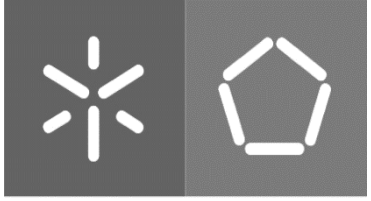
Universidade do Minho
Escola de Engenharia

José Alberto Moreira e Silva

Arm TrustZone: Evaluating the Diversity of the Memory Subsystem

Julho de 2019



Universidade do Minho
Escola de Engenharia

José Alberto Moreira e Silva

Arm TrustZone: Evaluating the Diversity of the Memory Subsystem

Dissertação de Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efectuado sob a orientação do
Professor Doutor Sandro Pinto

Julho de 2019

Acknowledgements

"How do you pick up the threads of an old life? How do you go on, when in your heart you begin to understand there is no going back?". Returning to the regular places and occupations after learning that life has drastically changed and will never go back to what it used to proved to be as difficult as described by Tolkien. As this work finishes, so does a chapter of life which was so far the hardest to let go, but thanks to a number of people, it can now be seen as a learning experience for what the future may hold, and for all of them I am thankful for impacting my life and getting me here. There are although some that I must refer to personally.

Firstly I'd like to thank my advisor, Sandro Pinto, for always being comprehensive and helping guide the elaboration of this work, always allowing me to put myself first and finish late instead of never completing this work. Secondly, a word of appreciation to José Martins, for sharing his knowledge and experience in the early stages of this work and helping getting over some initial humps in the road.

To all of the colleagues at the ESRG lab—Ailton, Ângelo, Francisco, Hugo, José, Nuno, Pedro, Ricardo and Sérgio—always ready to lend a helping hand whenever someone needed it, I'm grateful for growing and improving alongside such great people. To the three old friends that have known me for almost as long as myself, thank you for being present, specially during this last year, and may the weekend company last for as long as possible.

Lastly, I have to thank my family for everything they have done to help me get to where I am today. Specially to my parents and sisters, I am forever grateful for being the support system I needed to get through life so far. When all other things came crashing down, you were the ones making sure I did not go along with the rubble, and for that I will always love you all.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Abstract

The diversification of the embedded market has led the once single-purpose built embedded device to become a broader concept that can accommodate more general-purpose solutions, by widening its hardware and software resources. A huge diversity in system resources and requirements has boosted the investigation around virtualization technology, which is becoming prevalent in the embedded systems domain, allowing timing and spatial sharing of hardware and software resources between specialized subsystems. As strict timing demands imposed in real-time virtualized systems must be met, coupled with a small margin for the penalties incurred by conventional software-based virtualization, resort to hardware-assisted solutions has become indispensable.

Although not a virtualization but security-oriented technology, Arm TrustZone is seen by many as a reliable hardware-based virtualization alternative, with the low cost and high spread of TrustZone-enabled processors standing as strong arguments for its acceptance. But, since TrustZone only dictates the hardware infrastructure foundations, providing SoC designers with a range of components that can fulfil specific functions, several key-components and subsystems of this technology are implementation defined. This approach may hinder a system designer's work, as it may impair and make the portability of system software a lot more complicated.

As such, this thesis proposes to examine how different manufacturers choose to work with the TrustZone architecture, and how the changes introduced by this technology may affect the security and performance of TrustZone-assisted virtualization solutions, in order to scale back those major constraints. It identifies the main properties that impact the creation and execution of system software and points into what may be the most beneficial approaches for developing and using TrustZone-assisted hardware and software.

Resumo

A recente metamorfose na área dos sistemas embebidos transformou estes dispositivos, outrora concebidos com um único e simples propósito, num aglomerado de subsistemas prontos para integrar soluções mais flexíveis. Este aumento de recursos e de requisitos dos sistemas potenciou a investigação em soluções de virtualização dos mesmos, permitindo uma partilha simultânea de recursos de hardware e software entre os vários subsistemas. A proliferação destas soluções neste domínio, onde os tempos de execução têm de ser respeitados e a segurança é um ponto-chave, tem levado à adoção de técnicas de virtualização assistidas por hardware.

Uma tecnologia que tem vindo a ser utilizada para este fim é a Arm TrustZone, apesar de inicialmente ter sido desenvolvida como uma tecnologia de proteção, dado a sua maior presença em placas de médio e baixo custo quando comparada a outras tecnologias. Infelizmente, dado que a TrustZone apenas fornece diretrizes base sobre as quais os fabricantes podem contruir os seus sistemas, as especificações da tecnologia divergem de fabricante para fabricante, ou até entre produtos com a mesma origem. Aliada à geral escassez de informação sobre esta tecnologia, esta característica pode trazer problemas para a criação e portabilidade de software de sistema dependente desta tecnologia.

Como tal, a presente tese propõe examinar, de uma forma sistematizada, de que forma diferentes fabricantes escolhem implementar sistemas baseados na arquitetura TrustZone e em que medida as mudanças introduzidas por esta tecnologia podem afetar a segurança e desempenho de soluções de virtualização baseadas na mesma. São identificadas as principais características que podem influenciar a criação e execução de software de sistema e potenciais medidas para diminuir o seu impacto, assim como boas práticas a seguir no desenvolvimento na utilização de software e hardware baseados na TrustZone.

Contents

List of Figures

List of Tables

List of Listings

Glossary

AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
APU	Application Processing Unit
APU	Application Processor Unit
AXI	Advanced eXtensible Interface
COTS	Commercial of-the-shelf
CPU	Central Processing Unit
CSU	Central Security Unit
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory
FIQ	Fast Interrupt Request
FPGA	Field Programmable Gate Array
GIC	Generic Interrupt Controller
GPOS	General-Purpose Operating System
IoT	Internet of Things
IP	Intellectual Property
IPA	Intermediate Physical Address
IRQ	Interrupt Request
LTZVisor	Lightweight TrustZone-assisted Hypervisor
MMU	Memory Management Unit
NS	Non-secure

OCM On-chip Memory

OS Operating System

PA Physical Address

PL Programmable Logic

PR Partial Reconfiguration

PS Processing System

RTOS Real-Time Operating System

SMC Secure Monitor Call

SoC System on-Chip

SPSR Saved Program Status Register

SVC Supervisor Call

TCB Trusted Computing Base

TLB Translation Lookaside Buffer

TTBR Translation Table Base Register

TZ TrustZone

TZASC TrustZone Address Space Controller

TZMA TrustZone Memory Adapter

TZPC TrustZone Protection Controller

USB Universal Serial Bus

VA Virtual Address

VE Virtualization Extensions

VM Virtual Machine

VMCB Virtual Machine Control Block

VMM Virtual Machine Monitor

XMPU Xilinx Memory Protection Unit

XPPU Xilinx Peripheral Protection Unit

1. Introduction

The diversification of the embedded market, prompted by the expansion of industries such as consumer electronics and the more recent incursion of Internet-of-Things solutions, has led the once single-purpose built embedded device to become a broader concept. These platforms can now accommodate more general-purpose solutions by widening its hardware and software resources. Furthermore, coupling multiple subsystems in the same platform does not only reduce costs, by promoting resource sharing among components, but also increases performance, as the tightly coupled subsystems can communicate a lot faster.

As a way of consolidating the multiple isolated subsystems into the same hardware platform, virtualization technology is becoming prevalent in the embedded systems domain, allowing timing and spatial resource sharing between subsystems by enabling the implementation of heterogeneous software environments into the same platform. This trend can be seen in industries such as the automotive industry, where real-time, safety-critical functionalities are integrated with the infotainment environment (Kim et al., 2013, Lee et al., 2016, Reinhardt and Morgan, 2014), or the aerospace industry, where virtualization provides isolation for safety-critical components (Masmano et al., 2009, Pinto et al., 2016b). In both cases, a Real-time Operating System (RTOS) and a General Purpose Operating System (GPOS) are each encapsulated in what is called a Virtual Machine (VM), which has a virtualized view of the system resources. This resource abstraction allows monitoring software—a Virtual Machine Monitor (VMM)—to manage VM communication to system components and among each other, in order to guarantee timing and spatial isolation between VMs. As strict timing demands imposed in real-time virtualized systems must be met, coupled with a small margin for the penalties incurred by conventional software-based virtualization, resort to hardware-assisted solutions is imperative and becoming the norm.

Among existing commercial of-the-shelf (COTS) technologies, such as Intel's Virtualization Technology (VT), Arm Virtualization Extensions (VE) and Arm TrustZone (TZ), the latter is asserting itself in the virtualization realm. Although not a virtualization but security-oriented technology, TrustZone is seen by many as a reliable hardware-based virtualization alternative (Frenzel et al., 2010, Pinto et al., 2019, 2016a, 2017b, Sangorrin et al., 2010). The lower cost and higher spread of TrustZone-enabled processors, in comparison with VE-enabled processors, along with the fact that TrustZone is considered the only implementable hardware-based virtualization approach on non-VE Arm processors, are standing as strong arguments for this solution's acceptance. Moreover, due to the introduction of TrustZone technology in the new generation of Cortex-M processors (ARM, 2017b), the technology is poised to make its mark in the low-end sector (Pinto et al., 2019), revealing the prospect for virtualization in embedded devices that are more resource-constrained.

As System on Chip (SoC) manufacturers continue to develop TrustZone compliant devices, this technology is becoming ingrained in the embedded market (ARM, 2018c). But, since TrustZone only provides the hardware infrastructure foundations, allowing a SoC designer to choose from a range of components that can fulfil specific functions within the target environment (ARM, 2009b), it means that a lot of the technology is implementation defined. This approach may hinder a system designer's work, as it may impair and make the portability of system software (Operating Systems, Hypervisors) developed with this technology a lot more complicated since, for example, different target platforms may provide more or less options to control its memory subsystem, as well as lead to some design idiosyncrasies that should be accounted for (Zhang et al., 2016a). As such, it is important to examine how different manufacturers choose to work with the TrustZone architecture, in order to scale back those major constraints.

1.1 Motivation

This master's thesis was developed to help shed some light into the TrustZone world, mainly the memory subsystem, of which information is rather scarce and superficial. With the current existence of TrustZone-enabled virtualization solutions, it becomes relevant to take a step back

and evaluate how the implementation-defined facet of this technology may influence the development of these solutions, as well as answering some questions that are left pending by the available TrustZone literature. Developed within the Embedded System Research Group (ESRG) of the University of Minho, it takes advantage of the existence of an in-house TrustZone-assisted hypervisor, the LTZVisor, and the current porting efforts for its deployment in various development boards, to evaluate the diversity and intricacies of the TrustZone memory subsystem in different platforms.

1.2 Goals

The main goals of this thesis are to survey and identify the main differences in separate implementations of TrustZone-aware memory subsystems, and how these may impact the execution and porting of system software. Furthermore it intends to study the impact that some design choices of this technology may have on the security and performance of TrustZone-assisted virtualization environments.

To accomplish these goals, a smaller set of objectives can be laid out:

- Study of the Arm architecture and TrustZone-assisted virtualization, for a better contextualization of the research environment;
- Review and evaluation of different SoC implementations of TrustZone memory subsystems, identifying the main attributes whose variance most influences the development of system software;
- Review and evaluation of the TrustZone modifications to the cache architecture, its benefits and drawbacks to the implementation of virtualized environments.

After the initial study, the evaluation of the different approaches designers follow when building a system that complies with the TrustZone architecture will identify the advantages they bring over each other and the drawbacks associated with each one. The targeted platforms (Xilinx Zynq-7000, Xilinx Zynq Ultrascale +, RaspberryPi3, i.MX6) are representative of different price ranges

from big-name manufacturers/vendors in the embedded market (EETimes and Embedded.com, 2017), which provide a somewhat broad view of the embedded landscape.

1.3 Document Structure

Including this introductory chapter that contextualizes the motivation behind the work, this document is divided into six chapters. Chapter 2 presents the underlying concepts necessary to the development of this thesis, reviewing the virtualization concept, Arm's architectural features as well as the TrustZone extensions and their use in the deployment of virtualization solutions. Chapter 2 concludes by shortly reviewing some works that helped identify some of the questions this thesis aims to answer. Chapter 3 presents the hardware platforms that were subject to evaluation and the runtime environment, LTZVisor, that was deployed in those platforms to perform the review of the different implementations. Chapters 4 and 5 present the results of the surveying processes of the main memory subsystem of the selected boards and TrustZone's cache-level implementations, respectively. Chapter 6 displays the main summary of the results drawn in the two preceding chapters, bringing together the conclusions from both angles, providing a more holistic view of the impacts they may present.

2. Background and Related Work

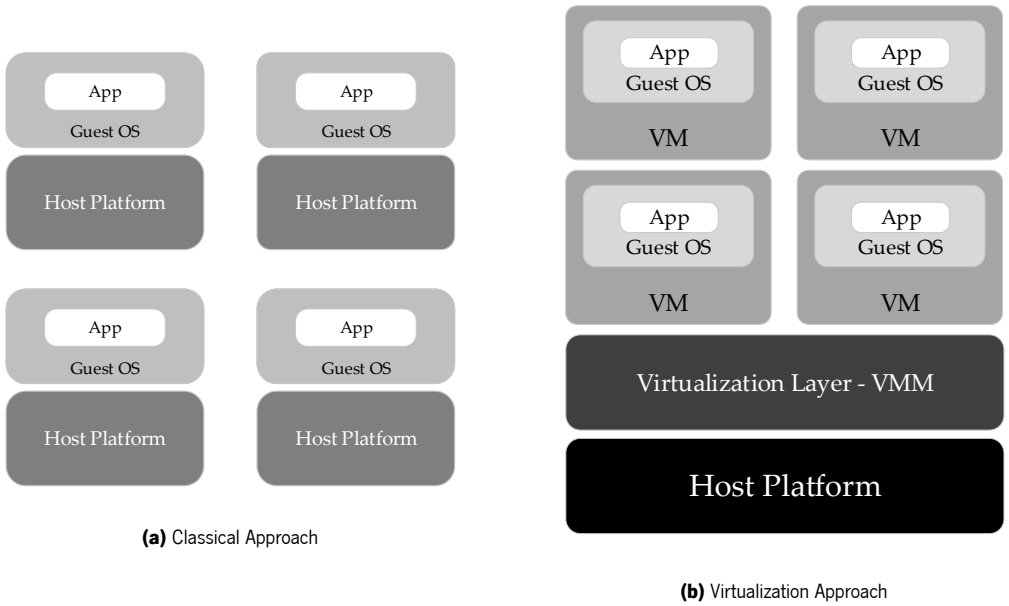
This chapter exposes the most pertinent subjects for the development of this thesis and reviews works that are closely related to the core of research. The concept of virtualization is introduced, followed by an overview of the Arm architecture and the Arm TrustZone extensions. Some works that provide an overview of the TrustZone technology are then reviewed for a deeper contextualization of this thesis focus.

2.1 Virtualization

Virtualization can be seen as the emulation of a host system into multiple isolated virtual machines (VMs) which mimic the behaviour of the host. A VMM, also known as an hypervisor, is responsible for configuring and maintaining the VM environment, running at the highest privilege level and multiplexing accesses from the VMs to the hosting system resources.

Despite being around since the 60's, it was not until a couple of decades ago that virtualization became a relevant topic in both research and commercial environments. Initially developed as a way to improve utilization of hardware resources on platforms that previously could only run a single application/Operating System at a time (Graziano, 2011), virtualization got out of focus for the research community when the availability of multitasking and multi-user OSes became more widespread. Its resurgence took place when, mainly in the increasingly connected world of desktop and server domains, challenges of resource management and isolation became a concern that prompted research for a better solution than fixing all scalability bottlenecks present in general purpose OSes (Kauer, 2014). Using virtualization to tackle this problem allows concurrent execution of multiple VMs to increase the utilization rates of underlying resources while also

providing VM isolation and encapsulation. Isolation ensures that concurrent VMs cannot tamper with each others data/execution flow while encapsulation allows saving of a VMs current state, making it possible to run at another time or even in a different host machine. Figure 2.1 depicts the architectural differences between the classical approach of single guest per host system and the virtualized approach in which a single host allows execution of multiple guest VMs.



(a) Classical Approach

(b) Virtualization Approach

Figure 2.1: System Stack Approaches

Traditionally, guests are unaware of their virtualization, in an approach called full-virtualization. This allows running unmodified OSes, which is crucial when dealing with OSes whose source code is not available but comes with a drawback in terms of performance and increased complexity. This is due to traditional trap-and-emulate or Dynamic Binary Translation techniques which require constant execution mode crosses and excessive use of memory resources. The trap-and-emulate techniques require that the hypervisor identifies when guests attempt to access critical registers or restricted memory locations, trapping those instructions, and then emulating the effect they would trigger, without actually executing them on the host, which greatly increases the hypervisor's workload.

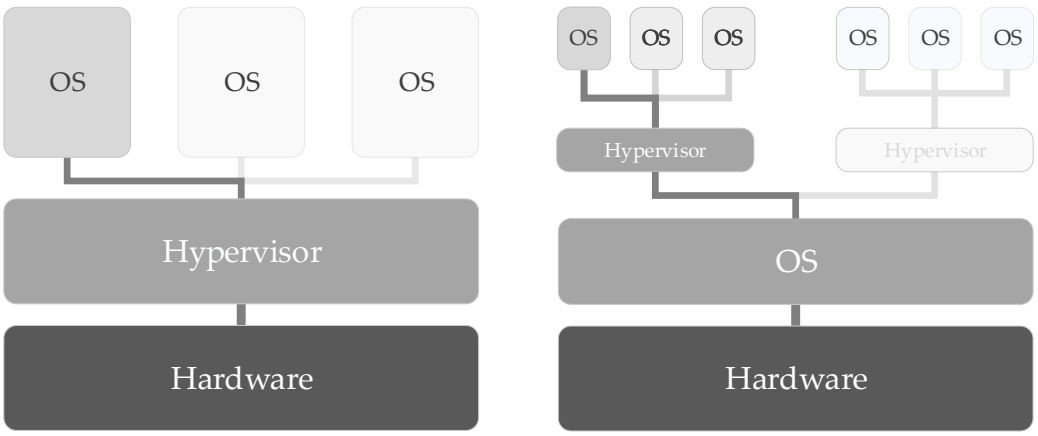
Differently to full-virtualization, para-virtualization relies on changing guest OSes source code, introducing hypercalls that implicitly invoke the hypervisor for execution of critical instructions. This comes at a high engineering cost and is not always a viable solution, given the availability of

guest source code, but, when applicable, greatly minimizes the performance overhead imposed by classical approaches by reducing the hypervisors' workload. This approach also eases the process of memory management in a virtualized system since the guests can be aware of their memory boundaries at compile time and, as such, can prevent accesses to memory locations that belong to other guests, maintaining the isolation between guest partitions.

Independently of the virtualization approach, it is possible to distinguish two types of hypervisors based on the position of the virtualization layer in the software stack (Figure 2.2):

- **Type-1**, or bare-metal Hypervisors, have direct access to the hardware layer and manage the execution permissions of every system component, mediating all hardware accesses. As a consequence, the performance degradation of guest OSes will only be influenced by the performance of the Hypervisor itself, making this type of Hypervisor more suited to systems with strict time constraints.

- **Type-2**, or hosted Hypervisors, run over a OS that is already executing rather than directly above the hardware layer. This type of VMM usually does not have permissions to access and perform any operation on the hardware directly, since those responsibilities usually rest in the system software that runs below the VMM, leading to lower performance ratings compared to type-1 Hypervisors.



(a) Type-1 Hypervisor **(b)** Type-2 Hypervisor

Figure 2.2: Hypervisor Classification

Fulfilling the growing need to consolidate multiple isolated subsystems into the same hardware platform in fields such as the automotive (Kim et al., 2013, Lee et al., 2016, Reinhardt and Morgan, 2014) and aerospace (Masmano et al., 2009, Pinto et al., 2016c) industries, this shift towards virtualized environments is also becoming prevalent in the embedded systems world, mainly through type-1 hypervisors. As the strict timing demands imposed in real-time virtualized systems must be met, coupled with a small margin for the penalties incurred by conventional software-based virtualization, resort to hardware-assisted solutions is becoming the norm. Among other things, virtualization hardware extensions—e.g. Arm's Virtualization Extensions (VE) (Varanasi and Heiser, 2011) or Intel's Virtualization Technology (Uhlig et al., 2005)—reduce overhead in context switch operations and eliminate some of the trapping of guest accesses by creating a new hypervisor privilege mode and introducing banking of sensitive registers between execution modes. Crucially, they also provide two-level address translation hardware support, speeding the translation from guest-virtual to host-physical address, which is essential when implementing a full-virtualization environment.

Although these hardware extensions are starting to make their way into the mid to high-end processors realm, propelled in a big way by the recent proliferation of Arm processors in this market, it is worth mentioning one other set of more widespread extensions developed by Arm. The Arm TrustZone extensions, initially deployed with security as its main target, implement similar register banking between privilege modes and have been shown to provide a virtualization solution in devices that do not possess hardware virtualization extensions (Frenzel et al., 2010, Pinto et al., 2017b, Sangorrin et al., 2010), enabling the integration of virtualized environments in a wider range of platforms.

2.2 Arm Architecture

In the past few years, Arm has helped build the technology that redefined industries ranging from mobile and consumer devices to networking and servers, cementing its position has a major force in the embedded market and in the Internet of Things (IoT) world. Arm licenses its processor architectures and intellectual property (IP) blocks—deployed in silicon to enhance/accelerate, for

example, security or graphics—to SoC manufacturers wishing to develop reliable and connected devices. Currently, Arm-based chips account for 39% of the "chips with processors" market share and growing, with 90% of the processors in mobile application processors or controllers for IoT devices being Arm-based (ARM, 2018c).

Previously known as Advanced RISC Machine, Arm represents a family of Reduced Instruction Set Computing (RISC) architectures which implements a small set of simple and general instructions, contrasting with a Complex Instruction Set Computing (CISC) architecture, which implements more complex and differentiated instructions. Traditionally, differences between these architectures allowed for lower cost, power consumption and heat dissipation values in RISC processors (Blem et al., 2013), fitting the needs of smaller, portable, battery-powered devices present in the embedded or mobile computing realms. Arm builds up on the base RISC 32-bit load/store architecture and enhances it by implementing features such as conditional execution of instructions to reduce branching and improve code density, auto incrementing and decrementing addressing modes that optimize program loops and load/store multiple data instructions to maximize data throughput (Goodacre and Sloss, 2005). All of these changes, along with the addiction of the reduced Thumb instruction set optimized for code density, with improved performance in narrower 16-bit memory systems, promoted the growth of Arm in the low to mid-end devices market, which is still prevalent to this day. Arm kept continually improving its architecture, adding new functionalities to the existing ones as the application processors requirements kept evolving, but maintaining its core characteristics like the different processor modes, general-purpose registers and program status registers. The most relevant additions to each version of the base architecture are depicted in Figure 2.3, such as the introduction of Single Instruction Multiple Data (SIMD) or the TrustZone security extensions and the virtualization extensions (ARM, 2005).

More recently, reflecting Arm's goal of becoming relevant in different markets which present distinct challenges and requirements, variations of the base architecture were introduced, to which Arm called architecture profiles. This differentiation of processor architectures allows manufacturers to comply with the current Size, Weight, Power and Cost (SWaP-C) requirements of modern

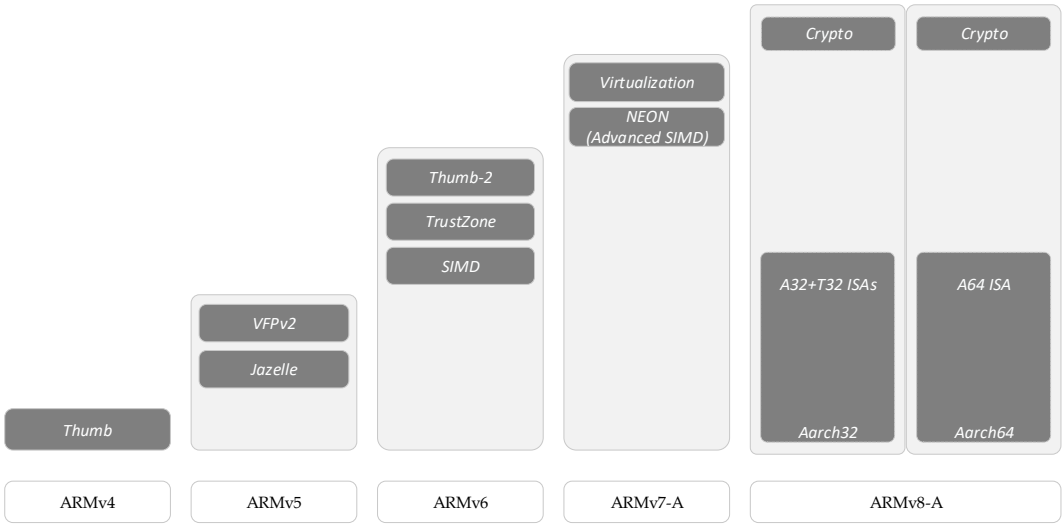


Figure 2.3: Arm architecture evolution

devices without compromising performance or determinism in systems where those are crucial. Chip manufacturers can choose between different profiles for the one which better fits their needs or even combine them, implementing a flexible system fit for a myriad of applications, without having the fear of consolidation issues between the modules. Currently, and since the unveiling of the Armv7 architecture, Arm provides three of these architectural variants and in-house processor core families called "Cortex" that are compliant with each of these profiles:

- Application profile: Armv7(8)-A processors, such as Cortex-A cores, implement a traditional Arm architecture and support a virtual memory system architecture based on a Memory Management Unit (MMU). This is the highest performance profile, influenced by multi-tasking OS system requirements, and is usually found in applications such as smartphones, tablets and digital televisions.

- Real-time profile: Armv7(8)-R processors, such as Cortex-R cores, implement a traditional Arm architecture similarly to the previous profile but support a protected memory system architecture based on an Memory Protection Unit (MPU) instead of the virtual memory approach followed by "-A" processors. As with the application processes, they provide high performance and throughput but with very precise timing and predictable interrupt latency, making them ideal for deeply embedded time-critical applications.

- Microcontroller profile: Armv7(8)-M processors, such as Cortex-M cores, implement a programmers' model designed for fast interrupt processing, with hardware stacking of registers and support for writing interrupt handlers in high-level languages. Intended for use in microcontrollers, they follow a minimalistic implementation when compared to the other profiles, which allows for low cost and low power values, characteristic of these types of devices.

As depicted above, these variants are available in both the Armv7 and the latest Armv8 architecture which, at the moment, are the two main Arm architectures in the market. Despite having an enormous presence in the mobile/smartphone industry (ARM, 2018c), the Armv8 architecture has not yet hit the embedded market in a meaningful way, where 64-bit devices are not very widespread, which boost the proliferation of Armv7 devices (EETimes and Embedded.com, 2017).

The following two subsections describe the Armv7-A and Armv8-A architectures, for a better understanding of the Arm architecture for application profiles, as these are the two architectures present in the boards subject to evaluation, in the form of Cortex-A9 and Cortex-A53 cores. In the context of this thesis, the covered subjects are narrowed to the processor modes, register organization, memory architecture and exception handling of the two architectures.

2.2.1 Armv7-A

Armv7, the seventh version of the 32-bit Arm architecture, was the first version to implement the distinction between three architectural profiles: the application profile, the real-time profile and the microcontroller profile. This section focuses on the application profile, Armv7-A, developed fo high-processing low power applications, such as smartphones or digital televisions, supporting a virtual memory system based on an MMU. Out of the three profiles, the application profile differs the least from the previous Armv6 architecture, building upon it while increasing capability and performance.

The following subsections present an overview of the Armv7-A architecture, in an implementation that includes both the TrustZone and virtualization extensions. Although the virtualization

extensions are not considered in the scope of this work, including them here allows a better comparison with the Armv8-A architecture.

2.2.1.1 Processor Modes

In an implementation that includes both the security and virtualization extensions, the Armv7-A architecture adds two processor modes to the seven present in Armv6 (ARM, 2005, 2012). The nine modes—User, System, Supervisor, Abort, Undefined, FIQ, IRQ, Hypervisor and Monitor—are split between privilege levels and security states. There are three privilege levels—PL0,PL1 and PL2— orthogonal to the security state, with the exception of the Hypervisor and Monitor modes which are assigned to the non-secure and secure states, respectively, as displayed in Table 2.1. The privilege levels refer to the permissions that software running at a certain level has over system resources in the current security state:

- PL0, where only User Mode runs, is considered the unprivileged level, allowing no access to system configuration registers. This is usually where OS applications that do not belong to the kernel run.

- PL1, where software has access to all system resources except for a few virtualization configuration registers added in the virtualization extensions. This is usually where OS software is executed. In this level, all modes except monitor mode can run in both security states.

- PL2, dedicated to the Hypervisor mode, is designed for virtual machine managing software and only runs in a non-secure state. Software executing at PL2 can perform all the operations available at PL1, plus some additional functionalities.

It is important to note that privilege level and security state are two independent concepts. The security extensions are further explained in section 2.3 but the main note to have present is that non-secure software cannot change configurations or control settings for secure operations, even in PL2, but secure software can meddle with non-secure operations. Permissions can be

Table 2.1: Armv7-A Processor Modes

Processor Mode	Privilege Level	Security State
User	PL0	Both
FIQ	PL1	Both
IRQ	PL1	Both
Supervisor	PL1	Both
Monitor	PL1	Secure
Abort	PL1	Both
Hypervisor	PL2	Non-secure
Undefined	PL1	Both
System	PL1	Both

assigned: at PL1, for accesses made by PL1 or PL0; at PL2 for non-secure accesses made by PL2 or non-secure accesses made by PL1 or PL0.

Changing processor mode can be achieved in three different ways: either privilege software explicitly changes the current processor mode or an exception is being handled or returned from (See Section 2.2.1.4). Changes from User mode can only occur by causing an exception, which is handled in a specific mode at a higher level, dependent of system configurations. Entering Hypervisor mode can only be achieved by taking an exception from non-secure PL0 or PL1 modes or returning from an exception being handled in Monitor mode. As for the PL1 modes, any of them can explicitly change the processor mode to another PL1 mode or to User mode.

2.2.1.2 Registers

Armv7-A maintains the general register layout of past versions, with thirteen general-purpose registers (R0-R12) and a Program Counter (PC) shared among all processor modes—except for the FIQ mode which has its own R8-R12 registers—, along with a Linker Register (LR), Stack Pointer (SP) and Saved Program Status Register (SPSR). These last three registers are banked between almost all processing modes, meaning that each mode has its own physical copy of the register, as displayed in (ARM, 2012, Figure 2.4), where the white cells represent the registers which are shared amongst all modes. When an exception is taken, the preferred return address is transferred to the banked LR register and the CPSR of the mode running when the exception was triggered is saved in the SPSR of the mode to which the exception was taken. This creates

	User	System	Hyp	Supervisor	Abort	Undefined	Monitor	IRQ	FIQ
R0	R0_usr								
R1	R1_usr								
R2	R2_usr								
R3	R3_usr								
R4	R4_usr								
R5	R5_usr								
R6	R6_usr								
R7	R7_usr								
R8	R8_usr								R8_fiq
R9	R9_usr								R9_fiq
R10	R10_usr								R10_fiq
R11	R11_usr								R11_fiq
R12	R12_usr								R12_fiq
SP	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
LR	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
PC	PC								
APSR	CPSR								
			SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq
			ELR_hyp						

Figure 2.4: Armv7-A Core Registers

an easier mechanism to return from exceptions, relieving the workload of the context restore.

These Program Status Registers (PSRs) store information such as the carry and overflow condition flags, the endianess, the execution state—Arm, Thumb or Jazelle—and the processor mode. Other system configurations regarding security and virtual memory are controlled through Arm's coprocessors, mainly CP15. These are only accessible by software running at PL1, with variable security permissions—registers can be secure access only, shared between both states, banked between both states and some have a configurable security access. This separation is further explored in section 2.3.

2.2.1.3 Memory Architecture

As previously mentioned, Armv7-A processors implement a Virtual Memory System Architecture (VMSA) based on a MMU, supporting the existence of level-1 (L1) and level-2 (L2) caches (ARM, 2012). In a VMSA implementation, all the addresses generated by the processor are Virtual Addresses (VA), enabling each kernel or user process to run in its own virtual memory space. This allows the concurrent execution of multiple processes atop of the same physical memory, through a process of address relocation which translates a virtual address issued by the processor into a physical address in main memory. In a classical implementation, a multitasking OS manages

each task's memory system view, switching the mappings when a task starts executing. In a virtualized system, in a two-level address translation implementation, another layer is added. So, the hypervisor manages each guest OS system view which then manages the tasks' translation regime. Figure 2.5 helps illustrate this concept—in a non-virtualized system, the guest physical address space would correspond to the real physical address space and the translation process would stop there.

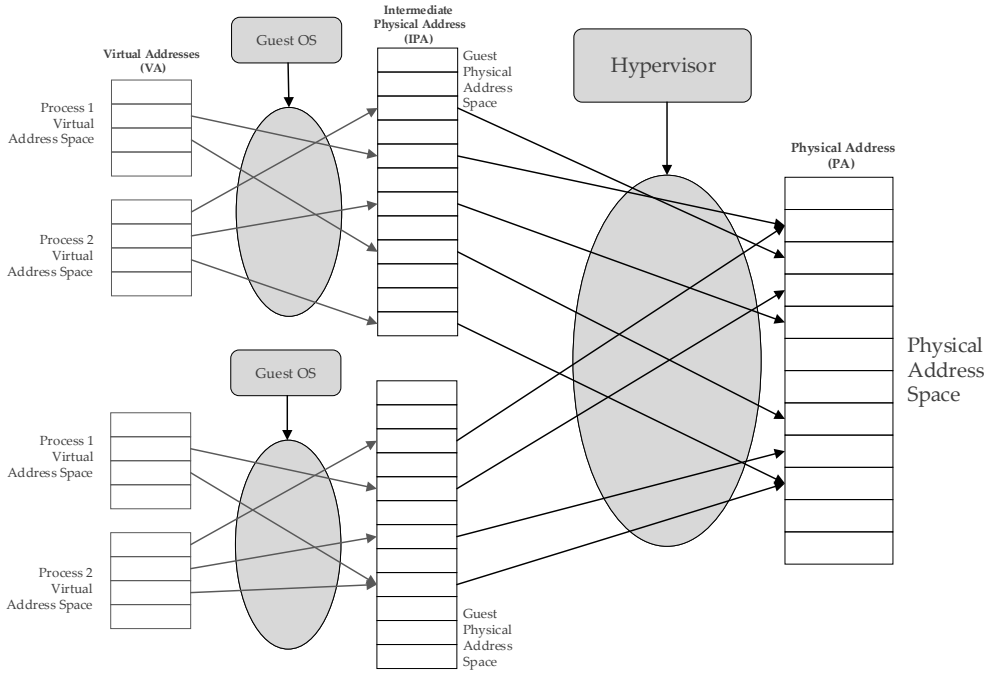


Figure 2.5: Two-level Address Translation

The unit responsible for this translation process between the virtual memory view and the real physical memory addresses is the MMU. Carried out by the MMU hardware, this process is transparent to the task issuing the virtual addresses, meaning that the task needs no knowledge of the underlying physical memory system mapping or the mappings used by other tasks. To translate these addresses, the MMU uses a set of translation tables which contain the mappings between virtual and physical addresses, along with memory access permissions and cache policies for each region of memory. When a virtual address is issued, the MMU uses the information in the translation tables to check if the access is valid and, if so, return the corresponding physical address, in a process called translation table walking. As these translation tables are usually stored

in main memory, which take some time to access, Arm implements a set of "translation caches", called Translation Lookaside Buffers (TLBs), which store recently executed translations. This access hierarchy is represented in (ARM, 2014, Figure 2.6). Note that the Caches block displayed in the figure may hold entries from the translation tables and not the result of the translation itself, which can only be cached in the TLBs.

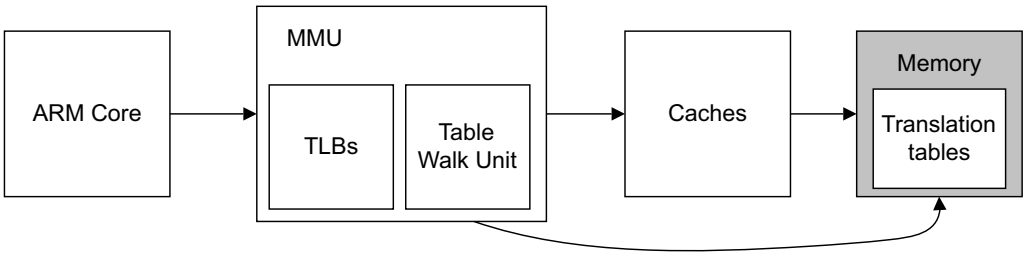


Figure 2.6: VMSA Access Hierarchy

Armv7-A, as a 32-bit architecture, spans a 4GB addressable physical space. Stage-1 translation tables, which translate virtual to physical addresses, split this 4GB into 4096 equal sized sections, each describing 1MB block of virtual memory space. Each of these 32-bit sized entries can describe pages of four different sizes: a supersection (16MB block of memory), a section (1MB block of memory), a large page (64KB block of memory) or a small page (4KB block of memory). The supersection and section entries define the base address of a contiguous 16 or 1MB section, respectively. On the other hand, small or large page entries point to a level 2 translation table, which allows subdividing a 1MB region into smaller pages, as displayed in Figure 2.7. The levels of translation tables should not be confused with the stages of translation where stage-1 refers to virtual to intermediate physical address translations (VA-IPA) and stage-2 to intermediate physical to physical address translations (IPA-PA).

Setting the base address of the stage-1 translations tables is done by writing to the Translation Table Base Register (TTBR), for stage-1 translations in every mode other than the hypervisor mode. For stage-1 translations from hypervisor mode, the Hypervisor Translation Table Base Register (HTTBR) is used. The stage-2 translation table base address is set by the Virtualization Translation Table Base Register (VTTBR), which is only accessible from hypervisor mode or monitor mode.

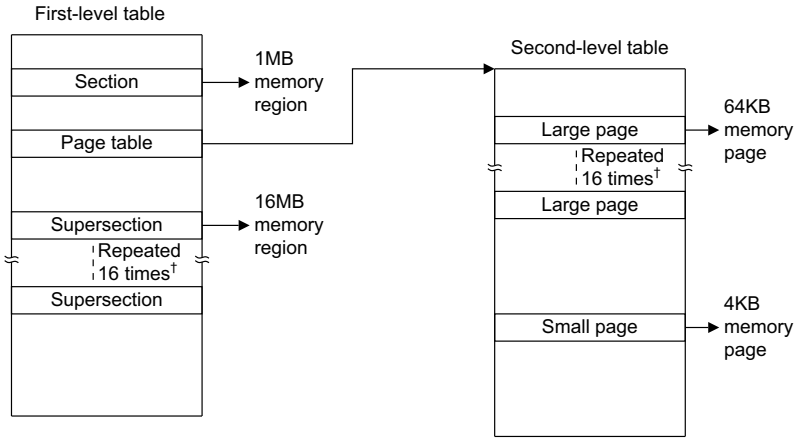


Figure 2.7: Armv7-A Page Table Layout

With the inclusion of the virtualization extensions, Armv7-A processors present four MMU interfaces, each with their own enable controls: secure PL1 and PL0 stage-1 MMU is controlled by the System Control Register MMU bit, SCTLR.M, in the secure copy of the register; non-secure PL1 and PL0 stage-1 MMU is controlled by SCTLR.M, in the non-secure copy of the register; non-secure PL1 and PL0 stage-2 MMU is controlled by the Hypervisor Configuration Register, HCR.VM, accessible only at PL2; non-secure PL2 stage-1 MMU is controlled by the Hypervisor System Control Register, HSCTLR.M, also only accessible at PL2. Disabling any of these instances means that the generated address remains unaltered but does not guarantee that the memory attributes or permissions are not set (ARM, 2012).

Latest versions of the Armv7-A architecture specification also introduced the Large Physical Address Extensions (LPAE), to support higher levels of device integration on a single memory and space under a single address space (ARM, 2011b). This allows the mapping of multiple 32-bit virtual address spaces into up to a 40-bit physical address space, solving some of the 4GB limitations present in 32-bit addressable systems.

2.2.1.4 Exception Handling

An exception causes the processor to suspend program execution to handle an event, such as an externally generated interrupt or an attempt to execute an undefined instruction. When an exception is taken in Armv7-A, the processor state is immediately preserved, before handling the exception, so that the original state can be restored upon exception return. This concept was

introduced in Section 2.2.1.1, where the linker register, stack pointer and SPSR were saved upon entering a new processing mode.

In this exception model, five types of exceptions are defined: reset, undefined instructions, system calls, memory aborts and interrupts. Arm takes these five types and implements nine different exceptions, incorporated in the security and virtualization extensions:

- Reset: When the reset input is asserted, the processor stops execution until that input is no longer driven. Execution is then resumed in secure Supervisor mode at the proper address defined in SCTLR.V.

- Undefined Instruction: Occurs whenever an instruction with undefined or invalid behavior tries to execute. This illegal execution can be the result of disabled coprocessor operations or unimplemented instructions for the current instruction set state.

- Supervisor Call (SVC): This instruction explicitly launches an exception that causes the processor to enter Supervisor mode. Typically, the SVC instruction is executed to request an operating system function.

- Secure Monitor Call (SMC): Part of the security extensions, this instruction causes the processor to enter Monitor mode.

- Hypervisor Call (HVC): Implemented with the virtualization extensions, the HVC instruction causes the processor to enter Hypervisor mode.

- Prefetch Abort: Triggered by a memory abort on an instruction fetch, usually generated by the MMU permission checking. By default this exception is taken to Abort mode but can be modified to change into Monitor or Hypervisor modes.

- Data Abort: Taken on a read or write memory access abort generated by the MMU or when a memory access external to the core generates an access error, such as a TrustZone security error. As with the Prefetch Abort, it can be handled in Abort, Monitor or Hypervisor modes.

- IRQ: Triggered by asserting an IRQ interrupt request input to the processor. Similar to the Abort exceptions, it is taken by default to the IRQ mode but can also be handled in Monitor or Hypervisor modes.

- FIQ: Alike the IRQ exception, it is triggered by asserting an IRQ interrupt request input to the processor and can also be handled in FIQ, Monitor or Hypervisor modes, depending on system configuration.

Out of all the exceptions mentioned above, it is important to note that the IRQ, FIQ and Data Abort exceptions can be masked using the Current Program Status Register. Masking them means that those exceptions will not be taken until they are no longer masked, which is useful when executing critical tasks that demand determinism and isolation. There is also another exception which has not been mentioned yet, the Hyp Trap. This is reserved for the virtualization extensions and is only taken when non-secure software running in any mode other than Hypervisor mode executes an instruction that traps in Hypervisor mode.

Handling exceptions in Armv7-A forces execution to an address that corresponds to the type of exception, referred to as the exception vector of that exception (ARM, 2012). These exception vectors are organized in four vector tables, assigned to the Hypervisor mode, the Monitor mode and the Secure and Non-secure states of the PL1. Each of these tables contains eight entries corresponding to specific exceptions' exception vector, organized as displayed in Table 2.2.

Table 2.2: Armv7-A Exception Vector Tables

Offset	Hypervisor	Monitor	Secure	Non-secure
0x00	—	—	Reset	—
0x04	Undefined Inst.	—	Undefined Inst.	Undefined Inst.
0x08	HVC	SMC	SVC	SVC
0x0C	Prefetch Abort	Prefetch Abort	Prefetch Abort	Prefetch Abort
0x10	Data Abort	Data Abort	Data Abort	Data Abort
0x14	Hyp Trap	—	—	—
0x18	IRQ	IRQ	IRQ	IRQ
0x1C	FIQ	FIQ	FIQ	FIQ

Usually a system programmer writes a branch instruction at these addresses that jumps to a more complex handler of the exception being taken. The base address to which these offsets

are relative to is saved in variations of the Vector Base Address Register (VBAR). The secure and normal worlds have banked copies of this register, accessible at PL1, while the HVBAR and the MVBAR respectively hold the Hypervisor and Monitor exception vector tables base address.

2.2.2 Armv8-A

Armv8 is the latest version of the Arm architecture and the first to implement a 64-bit architecture. The introduction of the Armv8-A architecture does not intend to tackle current limitations in application processors but to pave the way for designs that will support tomorrow's applications (ARM, 2017a). As previously mentioned, contemporary solutions in the embedded market still heavily rely on the Armv7-A architecture, boosted by the addition of the Large Physical Address Extensions which help solve the limitations of 4GB physical address spaces. As with all version of its architecture, Arm tries to maintain as much backwards compatibility possible, easing porting efforts to the new architectures. The two execution states implemented in Armv8-A are a reflection of this view:

- Aarch32 retains the Armv7 32-bit approach and maintains its definitions of privilege levels, allowing the execution of A32 or Thumb instruction sets.
- Aarch64 introduces the A64 instruction set with 64-bit general-purpose registers along with a new approach to privileged execution.

The following sections will focus on the innovations introduced in Armv8-A and, as such, are centered around the Aarch64 execution state. The division between these topics is homologous to the one made in the Armv7-A review, to facilitate a comparison between the two. As with section 2.2.1, matters such as the instruction set and interrupt management are not covered, as they are out of the scope for the development of this thesis.

2.2.2.1 Exception Levels

One of the biggest changes introduced in Armv8-A is the refactoring of the privileged execution model. When the virtualization extensions were introduced to Armv7-A, with the addiction of PL2,

separation of mode and privilege level became even more confusing than with the addition of monitor mode in TrustZone. As displayed in (ARM, 2015, Figure 2.8), monitor mode, used for switching between the secure and normal worlds, was the most privileged execution mode in Armv7-A but ran at the same privilege level as all other "privileged modes". Being placed in PL1 and with more privilege than PL2 was also counter-intuitive, not very clear from a numbering standpoint.

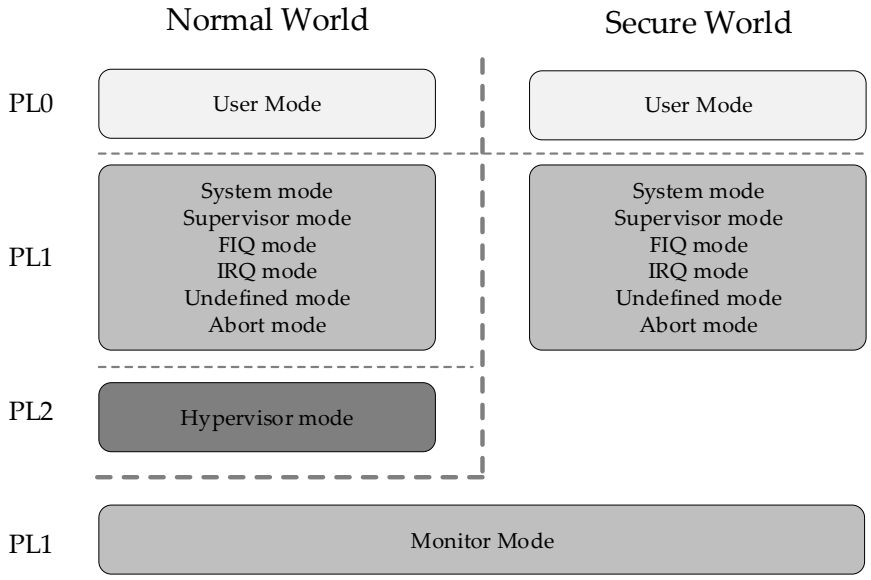


Figure 2.8: Armv7-A Privilege Levels

With the introduction of Armv8-A, monitor mode evolved into an exception level, EL3, with its own memory translation regime and interrupt vectors. This allowed for cleaner separation of code, making it easier to provide separate binary images for switching and service implementation (ARM, 2018b). This change into exception levels was also translated to the other processor modes, which disappear in Armv8-A, while the previous implemented privilege levels are translated into the new exception level model:

- EL0, where non-privileged software runs, usually refers to user-level applications of an OS.
- EL1, where OS kernel management and operations execute, in a privileged manner.
- EL2, reserved for the execution of an hypervisor, solely implemented in non-secure state.

- EL3, where the Secure Monitor now resides, in charge of switching between secure and normal worlds.

A typical software stack for systems deployed in Armv8-A processors is represented in (ARM, 2015, Figure 2.9), where the privilege hierarchy and separation is well defined and easily comprehended from an holistic point of view. As with Armv7-A, the now exception levels refer to the permissions that software running at a certain level has over system resources in the current security state. This applies to all ELs except the EL2 which can only access non-secure resources and the EL3 which can access all system resources at all time.

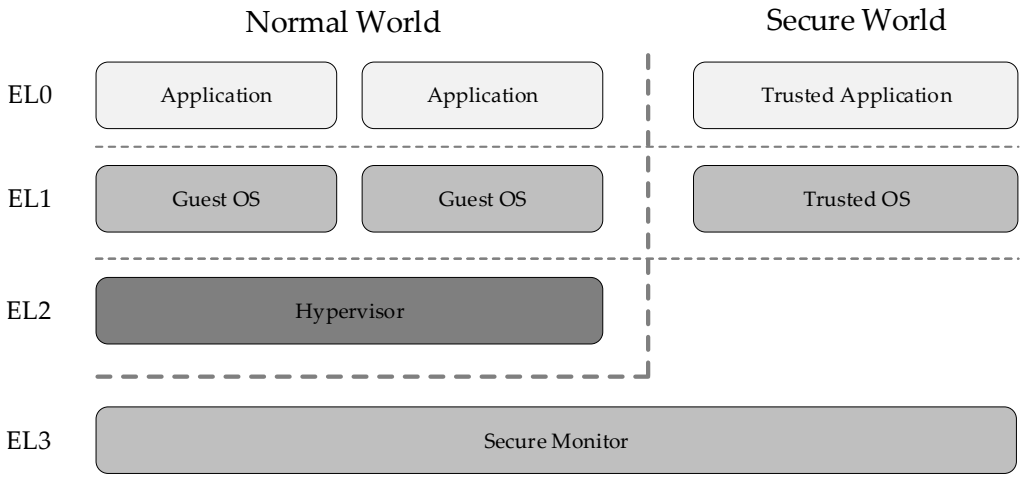


Figure 2.9: Armv8-A Exception Levels

Changes to the current exception level can be divided into two categories: raising to a higher privilege level or lowering to a less privileged level. Transitions to a higher exception level can only be achieved when handling an exception. These exceptions can either be the result of an internal or external event or through explicit execution of one of three system calls: a Supervisor Call (SVC), an Hypervisor Call (HVC) or a Secure Monitor Call (SMC). Supervisor calls are used by software running in EL0 to trigger a raise to EL1 and are usually executed by user applications wishing to use specific OS services. Hypervisor calls can be executed by software running in non-secure EL1, raising the exception level to EL2. Secure Monitor calls can be issued by software running at either EL1 or EL2, triggering a switch to EL3. SMC can also be trapped into EL2 when executed

from non-secure EL1 if the HCR_EL2.TSC bit is set. Transitions to a lower exception level are the result of executing an ERET instruction which restores the execution state according to the Saved Program Status Register (SPSR). As with Armv7-A, this register saves the current processor state—condition flags, executions state, exception level— when an exception is taken and, along with the linker register storing the address to return to, enables a quick context restoring process.

2.2.2.2 Registers

The shift to 64-bit processing and elimination of processor modes is reflected in the register organization changes introduced in Armv8-A. The new architecture is supported by thirty one 64-bit general purpose registers which are common to every exception level. In order to maintain compatibility with the Aarch32 execution mode, each of these registers can be accessed in one of two ways with the usage of a prefix. Using the "X" prefix gives access to the full 64-bit register while accesses executed with the "W" prefix can only affect the least significant thirty two bits of that register. As such, these registers range from X0-X30 or W0-W30, depending on the pretended address resolution. Out of the thirty general purpose registers, Arm's Application Binary Interface (ABI) for the 64-bit architecture defines X0-X7 as argument registers, doubling the amount of registers available for passing parameters in comparison to Armv7-A, reducing the need to spill parameters to the stack (ARM, 2017a, Figure 2.10). Registers X29 and X30 are also defined as the Frame Pointer register (FP) and Link Register (LR). None of these registers are banked between exception levels, which means that saving the general purpose register context between exception levels must be explicitly done by software.

Apart from these registers, Armv8-A also implement a set of "special registers" (ARM, 2017a, Figure 2.11). Two of them, the program counter (PC) and the zero register (XZR,WZR), are shared among all exception levels. The program counter in Armv8-A shifts away from the Armv7-A implementation, where it was considered a general purpose register, and is no longer accessible for explicit operations. The zero register is a register that reads as zero when used as a source register and discards the result when used as a destination register. Remaining special registers are split among the exception levels. Any of the Saved Program Status Registers (SPSR_ELx) can

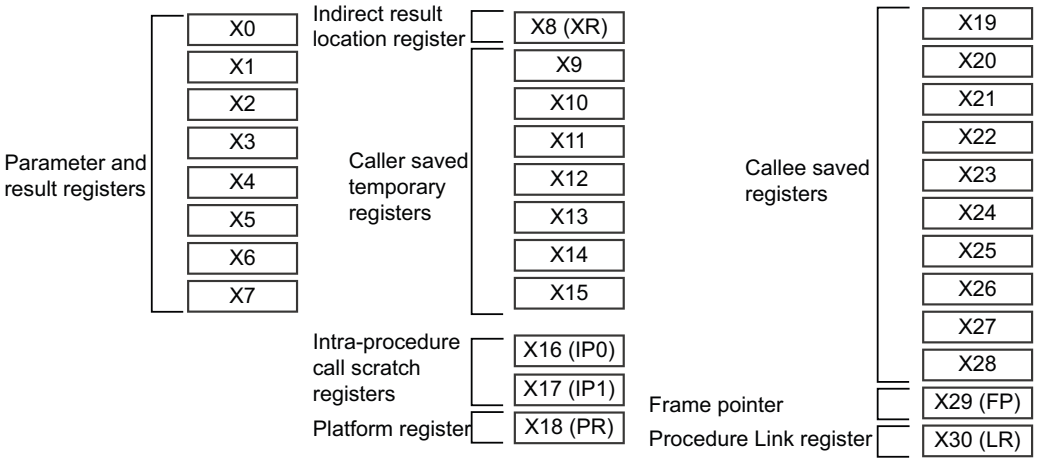


Figure 2.10: Armv8-A General-Purpose Registers

be accessed in any exception level equal or above the "x" level. For example, SPSR_EL1 can be accessed at EL1, EL2 and EL3 but not at EL0. EL0 does not possess a SPSR or a ELR since these registers are used to restore processor state when returning from an exceptions, which does not happen at EL0. As for the stack pointer, in any EL except EL0 both the current EL stack pointer or the SP_EL0 can be chosen as the stack pointer.

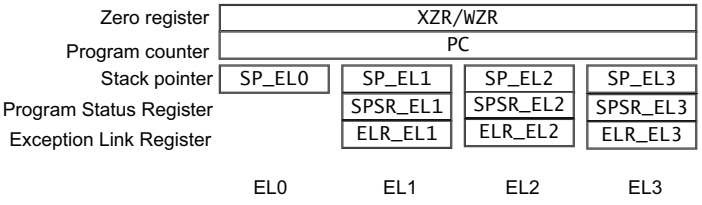


Figure 2.11: Armv8-A Special Registers

Regarding system configuration, which was previously controlled through the use of coprocessors, mainly CP15, Armv8-A implements different system registers throughout the execution levels that control various configurations. Akin to the special registers mentioned above, system register's names are extended with "_ELx" which indicates the lowest exception level at which a register can be accessed. Some registers only exist in certain exception levels, but are still extended with the same suffix, such as the SRC_EL3 register or virtualization registers tagged with the EL2 suffix. It is also important to note that even though a register may only differ in the

ELx suffix, functionalities of system registers assigned to lower exception levels do not have, most of the times, as many functionalities as the ones only accessible at higher exception levels.

2.2.2.3 Memory Architecture

The Virtual Memory System Architecture of an Armv8-A processor is, from a top-level viewpoint, similar to an Armv7-A processor implementation with virtualization and large physical address extensions, given the presence of two-level translation support and capability for physical address mapping above 4GB. Although implementing some different granule sizes and addressable bit depths, the main differences arise with the addition of the exception levels and in nomenclature standardization to minimize confusion.

In Armv7-A, monitor mode, which ran at PL1, had the same system view as any other processor mode running at PL1 in a secure state. The translation table base register (TTBR) used for stage-1 translations was banked between the secure and normal worlds but common to all processor modes. In Armv8-A, the base address for stage-1 translation tables are given by the TTBR0_EL1 or TTBR1_EL1 registers, which can be used by processes running at EL0 or EL1. Now supporting a 48-bit physical address memory map, selection between these two registers is based on the first sixteen bits of the 64-bit virtual address generated by the processor, which must be all 0s or 1s, otherwise triggering a fault (ARM, 2015). In AArch64 these registers are not banked between the secure and normal worlds and, as such, secure monitor code must manage tables for the both worlds, saving and restoring copies of TTBR0_EL1 and TTBR1_EL1. As for the stage-2 translations, which convert an intermediate physical address to a physical address, an extra set of tables is used, under control of the hypervisor. These only apply to non-secure EL0/EL1 accesses made by VMs managed by the VMM. The base register for the stage-2 translation table is specified in the Virtualization Translation Table Base Register (VTTBR0_EL2). The overall Armv8-A VMSA architecture is illustrated in (ARM, 2015, Figure 2.12), where the possible translation regime hierarchies are displayed along with the registers which configure the translation tables base addresses.

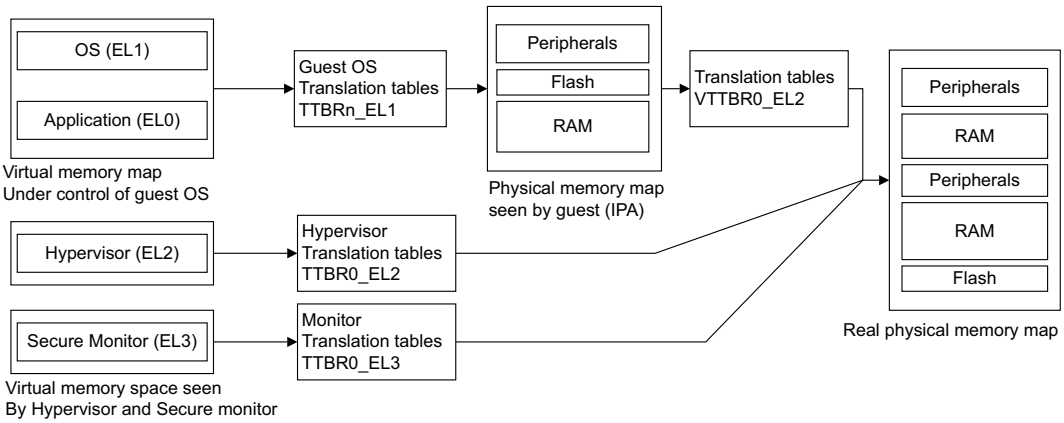


Figure 2.12: Armv8-A VMSA

Apart from stage-1 translations from EL0/EL1 or stage-2 translations, another two translation tables are implemented in Armv8-A. Hypervisor EL2 and secure monitor EL3 stage-1 translation tables are pointed to by TTBR0_EL2 and TTBR0_EL3, respectively. These tables provide the stage-1 translation regime for software running in the most privileged execution modes which always result in a virtual-to-physical address translation.

2.2.2.4 Exception Handling

Armv8-A maintains the logic behind exception handling mechanisms implemented in Armv7-A but, with the removal of processor modes and addition of the exception levels, taking exceptions no longer triggers a processor mode change but an exception level raise. As such, exception vector tables are implemented in all levels that can handle exceptions, i.e., all levels excluding EL0. Along with these changes, Armv8-A also reduced the differentiated exception types that can be handled. In the new model, exceptions can be the result of a reset signal, interrupts in the form of IRQs or FIQs, aborts, system calls or system errors (SErrors). These types are further condensed in each exception vector table where each entry can refer to a Synchronous exception, an IRQ, a FIQ or an SError. Synchronous exceptions include all system calls and the aborts explicitly resultant of executing an instruction, while SErrors are the result of asynchronous aborts, usually generated by faulty accesses to main memory. The reset exception is a special case which is only present in the highest implemented exception level and possesses its own

handling address, which is stored in the register RVBAR_ELx, where x is the highest implemented exception level.

Table 2.3 displays the exception vector table for a particular EL, where each entry spans one-hundred and twenty eight bytes (thirty two instructions). This presents a sizeable increase over the four byte entries present in Armv7-A which almost always represented some form of branch instruction to the actual exception handler. The space for thirty two instructions allows the immediate handling of exceptions but is also useful to determine the source of a synchronous exception or an SError. In these cases, the Exception Syndrome Register (ESR_ELx) is updated to indicate if the cause was for example a system call, a data abort or unallocated instruction execution.

Table 2.3: Armv8-A Exception Vector Tables

Offset	Exception Type	Description
0x000	Synchronous	Current EL with SP0
0x080	IRQ/vIRQ	
0x100	FIQ/vFIQ	
0x180	SError/vSError	
0x200	Synchronous	Current EL with SP0
0x280	IRQ/vIRQ	
0x300	FIQ/vFIQ	
0x380	SError/vSError	
0x400	Synchronous	Lower EL using AArch64
0x480	IRQ/vIRQ	
0x500	FIQ/vFIQ	
0x580	SError/vSError	
0x600	Synchronous	Lower EL using AArch32
0x680	IRQ/vIRQ	
0x700	FIQ/vFIQ	
0x780	SError/vSError	

As displayed above, the four previously mentioned entry types are mirrored in four sets which help identifying the execution state when the exception occurred, facilitating the access to the data of the faulting environment. Exceptions are taken to one of the subsets when: the exception was triggered on the current EL while using SP0; the exception was triggered on the current EL while using the SP of the corresponding EL (SPx); the exception was triggered on a lower EL which was running on the AArch64 execution state or the exception was triggered on a lower EL which

was running on the AArch32 execution state. With the four exception types available for each execution environment, the Armv8-A vector tables are comprised of sixteen entries, addressable by an offset relative to the table's base address, given by the VBAR_ELx register.

2.3 Arm TrustZone

TrustZone technology represents a set of hardware security extensions that enforce system-wide secure implementations of Arm-based SoCs by integrating protective measures into the Arm processor, bus fabric and system peripherals. This system-wide approach of integrated hardware components provides a framework for securing system architectures to be implemented with minimal impact on the cost of the end devices (ARM, 2009a, Pinto and Santos, 2019).

Arm introduced the TrustZone extensions in the Armv6Z update to the Armv6 architecture, and it has since become available in the application profile of Armv7 and Armv8 processors, with a recent extension to the microcontroller profile of the Armv8 architecture (Pinto et al., 2019). The security extensions represent Arm's attempt at creating a trusted environment that is not constrained to static configurations and allows isolated execution of third party software, in contrast to the traditional Trusted Platform Modules (TPMs) approach to security (Sabt et al., 2015). To accomplish this, TrustZone is implemented from the processor down, sort of turning a whole SoC into a programmable trusted platform module where any part of the system can be secured (ARM, 2009b).

2.3.1 TrustZone Architecture

The foundation behind TrustZone's hardware architecture can be seen as a special kind of virtualization approach that provides hardware support for memory, I/O and interrupt virtualization. At its core, TrustZone relies on the virtualization of a single physical core into two virtual cores, creating two completely separated execution environments: the secure world and the normal world. Security is enforced through isolation between both worlds, which is assured by preventing the normal world from accessing any resource assigned to the secure world and, on the other hand, allowing the secure world to have access to system resources regardless of their security state.

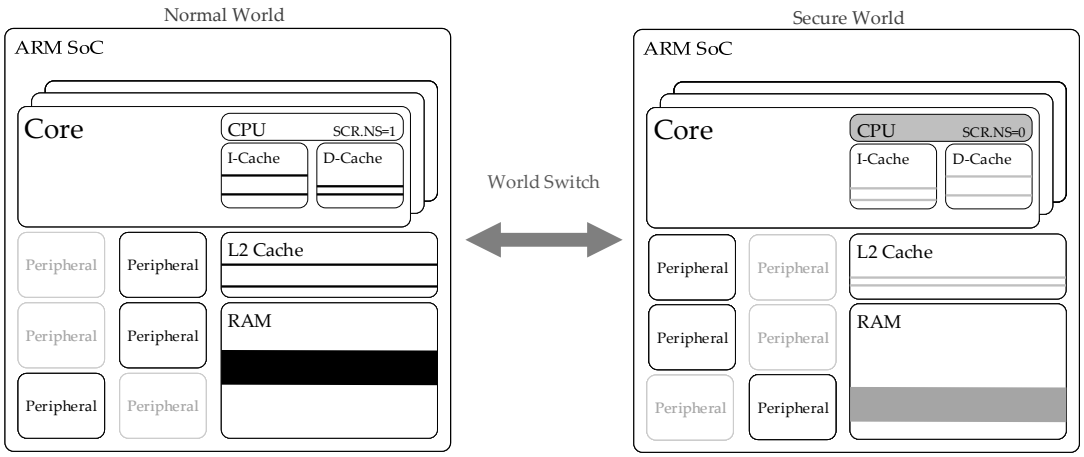


Figure 2.13: Arm TrustZone Architecture

This creates a security boundary that allows the containment of untrusted applications by limiting their access to system resources and, at the same time, allowing secure software to diagnose and maintain a healthy execution environment.

To manage execution between both worlds, TrustZone extensions add a new processor mode, monitor mode, which always runs in the secure world and at the most privileged processor execution level. To switch between worlds, monitor mode must set (or clear) the Non-Secure (NS) bit in the Secure Configuration Register (SCR), which controls the processor's current security state (Figure 2.13). In an implementation that only contains the security extensions, the NS bit dictates the security state of every processor mode excluding the monitor mode. Differently, when a processor implements the virtualization extensions, or in an Armv8-A implementation, the NS bit controls the security state for software running at PL0/PL1, or EL0/EL1. Software running at PL2/EL2 always runs in the normal world.

Entering monitor mode can be done either through the execution of a Secure Monitor Call (SMC) instruction or by configuring monitor mode to handle interrupt and abort exceptions triggered in other modes. This relies on the implementation of an extra exception vector table in monitor mode to complement the vector tables of the secure and normal worlds. The exception handling separation between worlds, specifically handling of FIQs and IRQs, is also supported by the General Interrupt Controller (GIC). In a TrustZone environment, the GIC can distinguish between secure or non-secure interrupts that can be independently routed to each world, while

also allowing for prioritization of secure interrupts over non-secure interrupts.

The virtualization of the CPU's system view is mainly supported by the register banking between both worlds and the distinction between secure and non-secure address spaces. Banked registers allow each world to have its own copy of a register without the need of saving its state each time a context switch occurs, further ensuring isolation between both worlds, complemented with the addition of system registers that manage security policies in the system, such as restricting non-secure accesses to system configurations. Likewise, the distinction between secure and non-secure address spaces allows a real physical address to separately exist in these two different environments. This is accomplished with the addition of a 33rd bit to the addresses issued by the processor, the NS bit, which indicates the type of access made to a physical memory location: if the NS bit is clear, the access was made from a secure state; if the NS bit is set, the transaction that accessed that address was made in a non-secure state. This address tagging is the result of the translation process between virtual-to-physical addresses performed by the MMU and is also present in the caches, where the same real physical address can exist in two different cache lines, one tagged as secure and another tagged as non-secure. This is useful for cache maintenance purposes but can introduce incoherence to the system (Zhang et al., 2016a), as discussed later on.

A TrustZone-aware Memory Management Unit (MMU) provides two distinct MMU interfaces, enabling each world to have a local set of virtual-to-physical memory address translation tables. This effectively means that the address space can be split into two subsets: a non-secure address space and a secure address space. Compliant with the TrustZone architecture, non-secure translation tables can only describe non-secure memory and, as such, only allow non-secure accesses to memory. Contrarily, secure tables can contain entries that are mapped to either a secure or non-secure address space, enabling secure world software to make secure or non-secure accesses.

Since the TrustZone architecture encompasses system wide extensions that go beyond the processor, connection between system components must also be TrustZone-extended. This can only be accomplished with a TrustZone-aware bus fabric that can propagate an access' security

state (NS bit) throughout the system. As such, the Advanced Microcontroller Bus Architecture (AMBA) Advanced Extensible Interface (AXI) is extended with a protection bit that signals if a transaction is secure or non-secure. With this extension, any component with an AXI interface can become TrustZone-aware. In the case of devices that use a different bus interface that is not extended with TrustZone capabilities, such as the Advanced Peripheral Bus (APB) used for low-power device communication, security access can be controlled through an AXI-to-APB bridge that provides an interface between the high-speed AXI domain and the low-power APB domain. This bridge includes a input signal for each peripheral connected to the APB bus which is used to determine if the peripheral is configured as secure or non-secure, rejecting non-secure transactions to devices configured as secure. These signals can be statically configured at synthesis time or, in conjunction with a TrustZone Protection Controller (TZPC), dynamically controlled at run-time.

As for securing On-chip memory (OCM) or external Dynamic Random Access Memory (DRAM), whose controllers are AXI-compliant, Arm provides two other system IP instances that enforce TrustZone extensions. The TrustZone Memory Adapter (TZMA) is a device that allows partitioning of an on-SoC static memory into two separate regions: a lower secure region and an upper non-secure region (ARM, 2006). The TZMA usually works in tandem with a TZPC which drives the secure region size input used by the TZMA to set the range of protected addresses. When securing external DDR memory or when there is a need to split a location into more than two regions, Arm provides the TrustZone Address Space Controller (TZASC). The TZASC can partition memory into several different secure and non-secure memory regions, using a programming interface which is only accessible from the secure side (ARM, 2008). Figure 2.14 shows how the security IP provided by TrustZone is usually organized in a traditional SoC implementation.

All of these system IPs are optional and implementation-specific components on the TrustZone specification, as Arm does not force their implementation in TrustZone aware systems-on-chip. Unlike the extensions to the processor architecture, SoC manufacturers can choose not to implement these logical blocks or implement their own versions of them. This is one of the factors that may hinder the deployment of system software reliant on particular characteristics of these

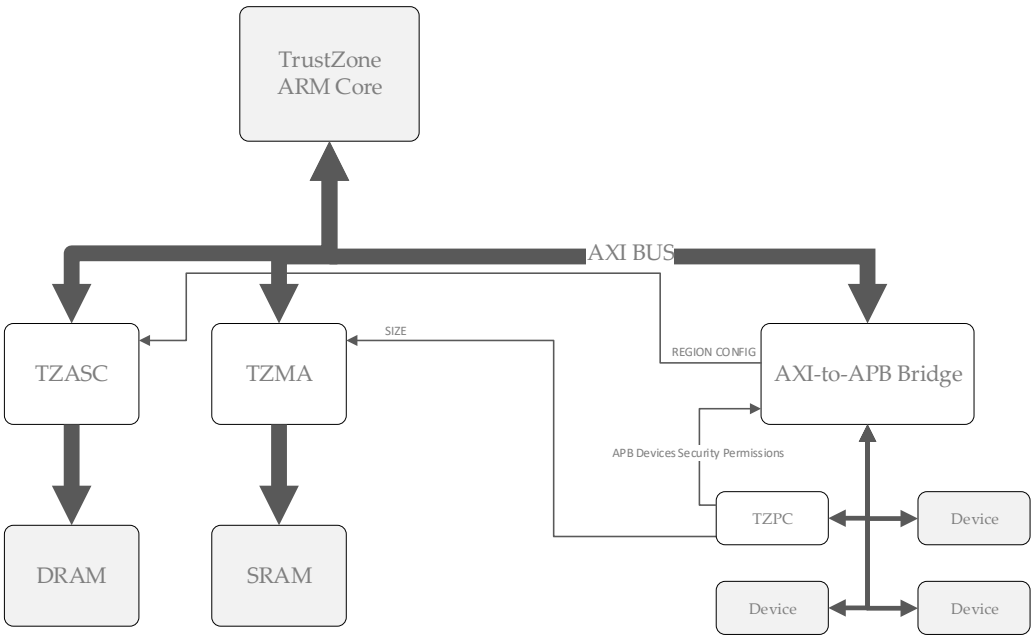


Figure 2.14: TrustZone-IP Example Layout

components, specially when talking about virtualized systems deployed in platforms which solely rely on the TrustZone extensions as their virtualization framework.

2.3.2 TrustZone-assisted Virtualization

As discussed in Section 2.1, hardware-assisted virtualization has been present in the embedded systems world for a while now, as hardware support for virtualization enables the deployment of virtualized solutions that meet the real-time requirements of embedded applications. Virtualization hardware extensions pertinent in the embedded world, such as Arm's Virtualization Extensions (Varanasi and Heiser, 2011) or MIPS VZ (Zampiva et al., 2015), add higher privilege execution modes where a hypervisor can run to manage execution of guest VMs. Removing the need to trap accesses to system configurations through register banking between hypervisor and guest modes, along with implementing support for two-level address translations and interrupt virtualization, these extensions greatly increase the performance of hardware-based virtualization when compared to classical approaches. Even so, the low availability of boards with these extensions in low-end to mid-range microprocessors prompted researchers to look at the Arm TrustZone

architecture (Figure 2.15) as a viable virtualization platform.

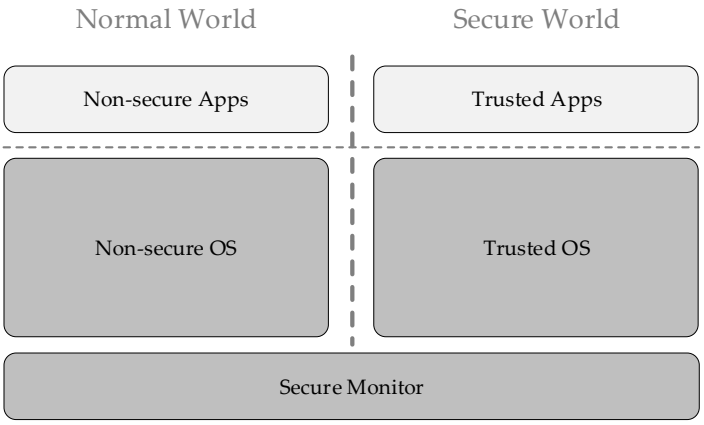


Figure 2.15: Arm TrustZone Software Stack

TrustZone does draw a lot of similarities with other hardware-based virtualization technologies and, even though initially developed as security extensions, it has proved to be a reliable virtualization framework in the embedded world. The addition of a higher privilege mode, above the unprivileged modes for guest OSes and user applications, which has control over the whole system and to which execution can be routed, along with the virtualization of interrupts between both worlds, mean that TrustZone is a suitable candidate for a hardware-assisted virtualization strategy. Except for the non-existence of hardware support for two-level translations, which hinders the implementation of full-virtualization environments, TrustZone possesses all the main characteristics of hardware-based virtualization systems, although isolation may be dependent of system resources. Since the sectioning of memory depends on the implementation of TrustZone peripherals, such as the TZASC and the TZMA, this also introduces the need for guests to be specifically compiled to run in their designed memory segments to maintain spatial isolation among them.

A traditional TrustZone-based virtualization solution relies on the time and spatial isolation features between the two worlds to create a VM environment in the normal world which is controlled by an hypervisor running in the secure world, in a single-guest approach (Figure 2.16). This is the simplest of the three main current state of the art approaches to TrustZone-assisted virtualization which also encompass dual-guest (Kim et al., 2013, Lucas et al., 2017, Pinto et al., 2017b, Sangorrin et al., 2010) and, more recently, multi-guest (Martins et al., 2017, Pinto et al., 2016b)

approaches. Despite not having many publicly documented solutions, single-guest approaches such as the ones presented by Frenzel et al. (Frenzel et al., 2010) or Douglas (Douglas, 2010) provide a proof of concept for TrustZone-assisted virtualization, implementing a small Trusted Computing Base (TCB) that secures underlying hardware resources from the execution of non-secure guests.

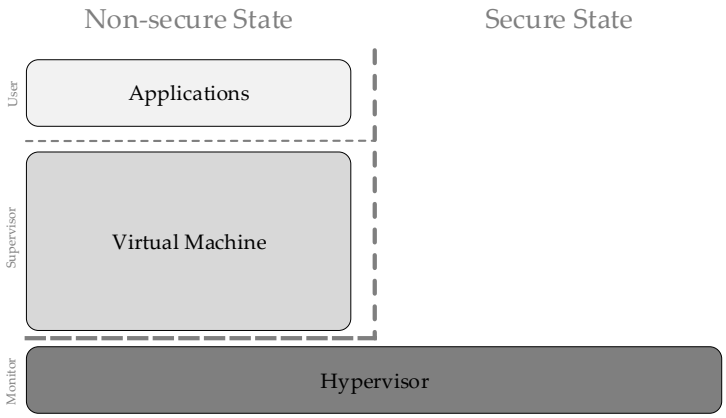


Figure 2.16: Single-Guest Virtualization Typology

The dual-guest approach (Figure 2.17) is the most common TrustZone-assisted virtualization configuration, as the implementation of two VM environments fits perfectly with the dual world separation, exploiting the advantages of register banking and isolation at cache level to optimize performance. This approach has shown to be a great match for mixed-criticality environments where, usually, a RTOS and a GPOS execute side by side in the same platform. Dual-guest environments work by creating a virtual machine (VM) environment in each world, whose execution is controlled by an hypervisor running in the secure monitor layer. To ensure the timing requirements of the RTOS, this operating system runs on the secure VM while the GPOS is deployed under the non-secure VM environment. Timing isolation between guests is usually enforced by an asymmetric scheduling policy that only allows the GPOS to run when the RTOS is in an idle state. Furthermore, the GIC is usually set to configure non-secure interrupts to be handled as IRQs and secure interrupts as FIQs. In terms of memory and device partitioning, this process is usually configured at boot time, which imposes slight changes in the RTOS source code, to prevent mismatches with the hypervisor configurations. As for the cache and MMU management,

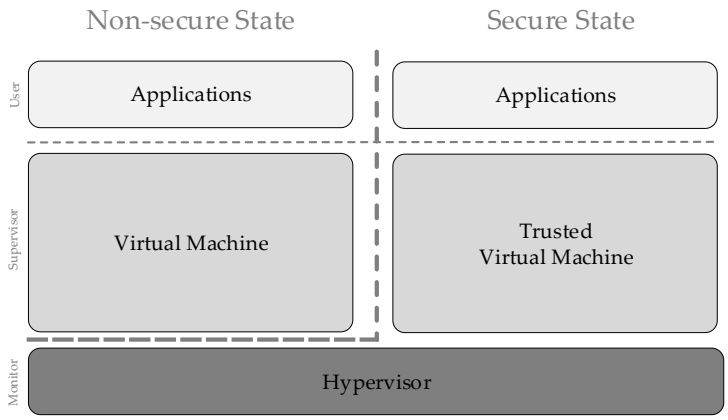


Figure 2.17: Dual-Guest Virtualization Typology

given that the two worlds posses different views of the memory subsystem, the need for context switch operations to enforce isolation is removed, as the separation is intrinsic to the cached accesses.

The more prominent dual-guest TrustZone-based hypervisors include:

- **SafeG** (Safe Gate), which presents an open-source solution to the dual-guest TrustZone-assisted virtualization approach. SafeG supports a health monitoring mechanism, a secure device sharing mechanism, and a cyclic and priority-based integrated scheduling policy (Sangorrin et al., 2010).

- **SASP** or Secure Automotive Software Platform, developed by the Korea University in conjunction with the Hyunday Motor Company, implements a dual-OS, RTOS and GPOS, configuration with focus on secure device access, providing isolation between a control system and an infotainment system in automotive vehicles (Kim et al., 2013).

- **LTZVisor**, a Lightweight TrustZone-assisted Hypervisor, is also an open-source dual-guest solution, initially supporting the Armv7-A and recently extending to the Armv8-A and Armv8-M architectures (Pinto et al., 2017b). LTZVisor also introduced the LTZVisor-AMP architecture (Pinto et al., 2017a), which implements support for a supervised asymmetric multi-processing configuration: one core runs in the secure world and hosts the secure software

(LTZVisor and RTOS), while the other core runs in the normal world and hosts the non-secure software (GPOS).

- **VOSYSmonitor**, developed by Virtual Open Systems, also implements a dual-OS virtualization but, unlike the previous cases, was implemented with the Armv8-A architecture as its target (Lucas et al., 2017). VOSYSmonitor leverages the virtualization extensions of the Armv8-A, allowing the execution of a non-secure hypervisor separated from the secure monitor layer where VOSYSmonitor runs.

More recently, researchers have tackled the main limitation behind TrustZone virtualization mechanisms in regards to scalability (in terms of number of supported guests) which prevented many authors from viewing TrustZone as a relevant virtualization solution (Pinto and Santos, 2019). Hypervisors like the RTZVisor (Pinto et al., 2016b) and its successor, the μRTZVisor (Martins et al., 2017), present an architecture that allows execution of multiple guest OSes, i.e., more than two guests. As depicted in Figure 2.18, multi-guest approaches maintain the hypervisor in monitor mode but delegate the execution of guests to the non-secure side, while the inactive guests are kept in the secure world. This relies on the implementation of a dynamic TZASC-

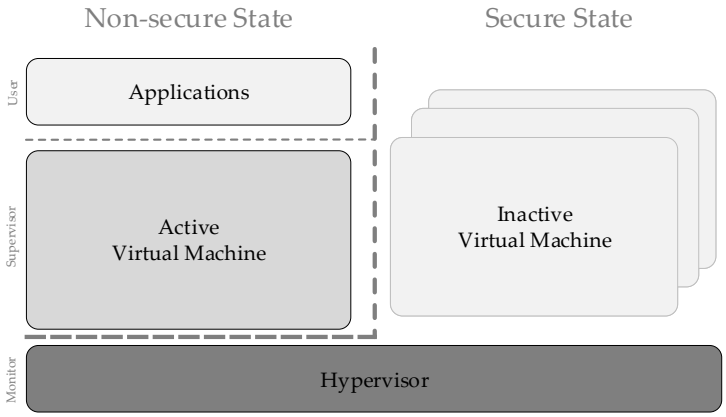


Figure 2.18: Multi-Guest Virtualization Typology

like controller so that only the active guest memory segment is kept in the non-secure side, while inactive guests are safeguarded in secure segments. This guarantees the isolation among guests, albeit with the caveat of requiring more cache and MMU maintenance operations at

context switches, given that the non-secure MMU and cache interfaces are shared between the guest partitions. Even so, this scalability issue remains prevalent given the variability of sectioning granularity provided by the TZASC-like components, which limits the number and size of guests, as further discussed in Section 4.

Given all these characteristics, it comes to no surprise that the dual-guest approach is the most widely environment for TrustZone-assisted virtualization. It is also relevant to note that solutions based on the TrustZone extensions mainly target the Armv7-A architecture. This is mostly due to the fact that there are still many industry platforms based on the Cortex-A9 processor architecture, which does not implement Arm's virtualization extensions, making TrustZone the only hardware-assisted virtualization option on these hardware platforms.

2.4 Related Work

Although there are some existing words focused on TrustZone-assisted virtualization or on the implementation of Trusted Execution Environments (TEEs) based on these extensions (Pinto et al., 2017, Pinto and Santos, 2019), public studies focused on the intrinsics of the TrustZone architecture are scarce. This limits the available knowledge of TrustZone's potentialities and limitations, which is detrimental to the development of solutions based on this technology, as well restricting discussion to possible improvements and/or adaptations to the extensions that can better fit the needs of system developers looking to implement TrustZone-based solutions. Even so, a few recent works have helped uncover some of the vulnerabilities of this technology and a couple of studies started painting a picture of the current TrustZone status and of what may be its use for future applications.

In CacheKit (Zhang et al., 2016a), Zhang et al. demonstrated that the TrustZone address tagging in cache lines can be a liability to the security of the system. This characteristic that allows the same physical address to coexist in two different cache lines, which is very convenient for performance, ends up breaking one of TrustZone's fundamentals if not handled properly. With CacheKit, Zhang shows that secure world cannot access cache lines tagged as non-secure, which become invisible to a secure state processor, allowing nefarious code (such as a rootkit) to be

hidden in the normal world cache, remaining undetected to conventional integrity system checking mechanisms (Figure 2.19). Even so, as further discussed in section 5, this exploitation is only viable when secure software has a memory view that does not reflect the true security state of the different memory segments.

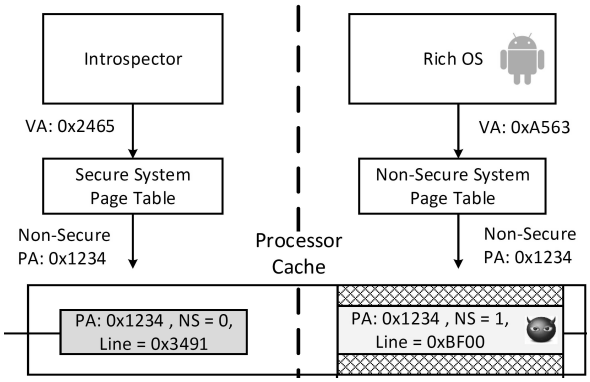


Figure 2.19: CacheKit Architecture

Continuing the exploitation of this cache duality, in TruSpy (Zhang et al., 2016b) Zhang once again proved that the separation between non-secure and secure cache lines in the same physical cache block can be used as a security exploit in TrustZone-enabled platforms. In this work, a type of cache-based attack called prime+probe is used by normal world software to extract an AES key used by secure world to carry out encryption services. The authors rely on the fact that even though non-secure lines cannot access secure ones, a non-secure cache fill can evict a secure line. This allows software running in normal world to track the cache utilization patters of the secure world, which can enable malicious software in the normal world to retrieve sensitive information stored in secure locations.

More recently, Benhani et al. (Benhani et al., 2017) analysed hybrid systems that combine powerful processing systems with reprogrammable FPGA fabric and pointed some exploitable weaknesses in the propagation of TrustZone between the two subsystems. Implementing small changes to the programmable logic, they were able to, among other exploits, circumvent secure access restrictions and create denial of service (DoS) attacks to the system by relying on the fact that security propagation to the PL is solely based on the AWPROT/ARPROT AXI signals, which

they could overwrite based on the target exploit. It is worth mentioning that these exploits would require that an attacker has access to the register-transfer level (RTL) code of the victim platform.

In a different panorama, a couple of works have been recently published with the intent to provide a holistic analysis of the TrustZone technology and its applications. These are mostly the result of reviews to the existing TrustZone-related literature (Ngabonziza et al., 2016), aiming to provide a context to the realm of everything TrustZone, hitting on architectural features and use cases, in a comprehensive manner that does not delve deep into specification but still allows for the identification of current flaws and future directions.

Very recently, Pinto and Santos published a highly comprehensive survey of the TrustZone technology, conveniently named Demystifying Arm Trustzone: A Comprehensive Survey (Pinto and Santos, 2019). This work presents a very thorough review of everything TrustZone, covering the past, present and foreseeable future of TrustZone's hardware and software, providing a very detailed picture of this technology's potentialities and vulnerabilities. This work starts by introducing the TrustZone architecture and its different facets in the application and microcontroller processor profiles, also identifying the variety and market placement of TrustZone-enabled platforms. Afterwards, the authors present the main characteristics that make TrustZone a viable solution for the implementation of various Trusted Execution Environments and virtualization solutions, always with the contextualization of current TrustZone-based solutions that are prevalent in either the academic or industrial worlds. Finally, the main vulnerabilities and security issues that have been identified by different authors are presented, before the authors reflect on the future directions of the technology and of TrustZone research. As with TrustZone Explained, this work was published with the intent of helping researchers and developers get familiarized with and encourage further exploration of the TrustZone technology.

3. Platform and Tools

This chapter provides an overview of the hardware platforms and software environment used in this evaluation. The evaluated TrustZone enabled platforms are briefly introduced, followed by the software framework on which the conducted tests were based, with the relevant nuances introduced in the version running in each platform. The TrustZone memory subsystem of the selected boards is not covered at this point as it is detailed and discussed in the following chapter.

3.1 TrustZone-enabled platforms

As TrustZone starts playing a major role in securing IoT devices, given the hegemony of Arm processors in this field, the number of TrustZone-enabled platforms is naturally increasing. In a recent thorough TrustZone survey, Pinto and Santos (Pinto and Santos, 2019) pinpointed the available range of TrustZone-enabled boards.

Regarding mid, high-end platforms (Cortex-A), the Xilinx platforms based on the Zynq-7000 SoC, or the more recent 64-bit UltraScale+, are usually the most attractive when starting to work with TrustZone, given that Xilinx provides vast technical documentation on the subject. NXP is another manufacturer whose reasonably-priced boards are well documented, being widely used in academic projects (Guan et al., 2017, Yalew et al., 2017, Zhang et al., 2016a). At the lowest price point in application devices, the RaspberryPi 3 has also been used in some projects (Liu and Srivastava, 2017), despite not providing many TrustZone-related resources. Aiming at a more industrial environment, companies like Nvidia and Renesas also present some platforms such as the Jetson TX2 DevKit and the R-Car Starter Kit, but choose not to disclose the details of their implementation.

In the low-end devices' sector, since the introduction of TrustZone to Cortex-M is relatively new (late 2016), TrustZone-enabled platforms are scarce. Nuvoton has recently presented the NuMicro M2351, based on the Cortex-M23 processor. Arm also came out with a development board, the Musca-A1, based on the Cortex-M33, although not yet commercially available.

Given the availability of boards and support documentation, the selected boards for this work were Xilinx's ZYBO trainer board and the ZCU102 evaluation board, aswell as NXP's MCIMX6DL-SABRE and RaspberryPi's RasperryPi 3B.

3.1.1 ZYBO Zynq-7000

The ZYBO (ZYnq BOard) is an entry level development platform manufactured by Xilinx, based on the Zynq-7000 SoC family. As with all Zynq-7000 products, it incorporates a dual core 32-bit Arm Cortex-A9 processor along with FPGA–based programmable logic in the same device (Xilinx, 2016a). Given the flexibility provided by the FPGA, in conjunction with the diverse I/O resources present in the different Zynq-7000 devices, this product range is suited for a wide set of applications in distinct fields such as automotive, industrial, medical or video and security equipment. As displayed in (Xilinx, 2016a, Figure 3.1), Xilinx's design divides these Zynq-7000 devices into two main subsystems—the Processing System (PS) and the Programmable Logic (PL)—on separate power domains.

The PS encloses memory and I/O interfaces, an Advanced Microcontroller Bus Architecture (AMBA) interconnect that enables communication between all the PS components—and with the PL—and the Application Processor Unit (APU). The latter contains the dual core Arm Cortex-A9 MPCore processor, a 32-bit multicore processor that implements the Armv7-A architecture, which is the main control block of the system. Each of its cores has associated 32KB data and instruction caches, that establish the first memory level (L1), and an MMU that enables virtual-to-physical address translations. Below this first level of memory is the 512KB unified cache and then the 256KB of On-chip Memory (OCM), kept coherent by the Snoop Controller, which connects to the Accelerator Coherency Port (ACP) for faster communication between processor and PL.

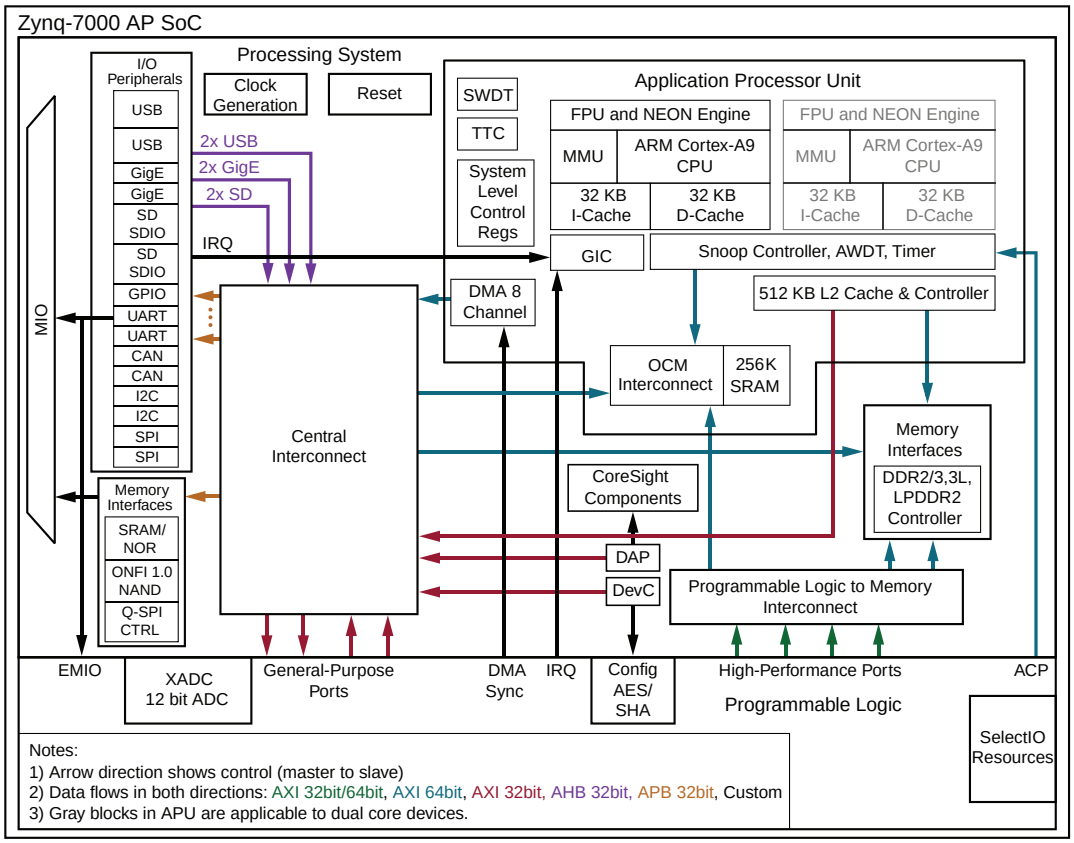


Figure 3.1: Zynq-7000 SoC Block Diagram

The PL, derived from Xilinx's FPGA technology, makes use of many types of resources such as configurable logic blocks (CLBs) and DSP slices to implement hardware accelerators that boost the systems' functionality. The PL is configured either during the boot process or later on by the PS, since the processors in the PS always boot first, allowing for a software-driven PL configuration. The PS can even reconfigure the PL, through a process called partial reconfiguration (PR). These options bring greater flexibility to the system, allowing for adjustments in the algorithms implemented in FPGA fabric, as well as in power management since the PL can be shutdown without affecting the PS.

In terms of connectivity, ZYBO provides various standard external interface peripherals—USB2.0, SD/SDIO, UART, SPI, I2C, Ethernet—, general-purpose I/O (GPIO) in the form of push-buttons, slide switches and leds, as well as media interfaces through audio input/output jacks and HDMI or VGA ports. An external 512MB of DDR3 memory are connected to the board through the dynamic memory controller, considerably expanding the board's usability for larger applications

where the small OCM would be lacking in terms of size.

As for security, besides allowing a secure boot process that loads authenticated and encripted PS images and PL bitstreams, ZYBO relies on its TrustZone features to enable developers to safeguard their systems. The secure boot process always initiates in a 128KB BootROM that isn't accessible to user applications and through a series of decryption operations determines if the boot images are secure or not. After the system boots, security is managed trough a selection of programmable TrustZone registers. All I/O devices mentioned in the previous paragraph can be configured as secure or non-secure by the developer at runtime. Memory regions, as described in the following chapter, can also be configured as secure or non-secure, but memory controllers are always set as secure.

Given that LTZVisor was originally tailored for Zynq-7000 platforms, ZYBO is both the starting point for this analysis and the chosen platform to test most of the MMU/cache scenarios, since the most stable version of LTZVisor will be the one running on these platforms.

3.1.2 MCIMX6DL-SABRE

Based on NXP's i.MX family of application processors, more specifically the i.MX6, the Smart Application Blueprint for Rapid Engineering (SABRE) boards are targeted towards smart devices that may require intelligent displays, connectivity and low power consumption. The i.MX processors are NXP's line of "multimedia application processors", optimized for the lowest power consumption while delivering high-performance processing capabilities. These characteristics make i.MX6 devices suitable for applications such as driver information systems, portable medic devices and smart mobile devices with media-centric purposes (NXP, 2017).

SABRE platforms come equipped with either an i.MX 6Quad applications processor or an i.MX 6DualLite applications processor, based on the Arm Cortex-A9 MPCore—i.MX 6DualLite supports a dual core Arm Cortex-A9 while i.MX 6Quad supports a quad core Arm Cortex-A9 configuration. MCIMX6DL is a DualLite platform and, as such, implements a dual core Arm Cortex-A9 MPCore configuration, similarly to the ZYBO board introduced previously. As with ZYBO, each CPU includes a 32KB L1 instruction cache and a 32KB L1 Data cache, with a 512KB unified instruction and data

cache shared between both cores. A top-level diagram of the i.MX6 DualLite SoC is presented in (NXP, 2017, Figure 3.2), displaying the main subsystems present in the chip and their connectivity.

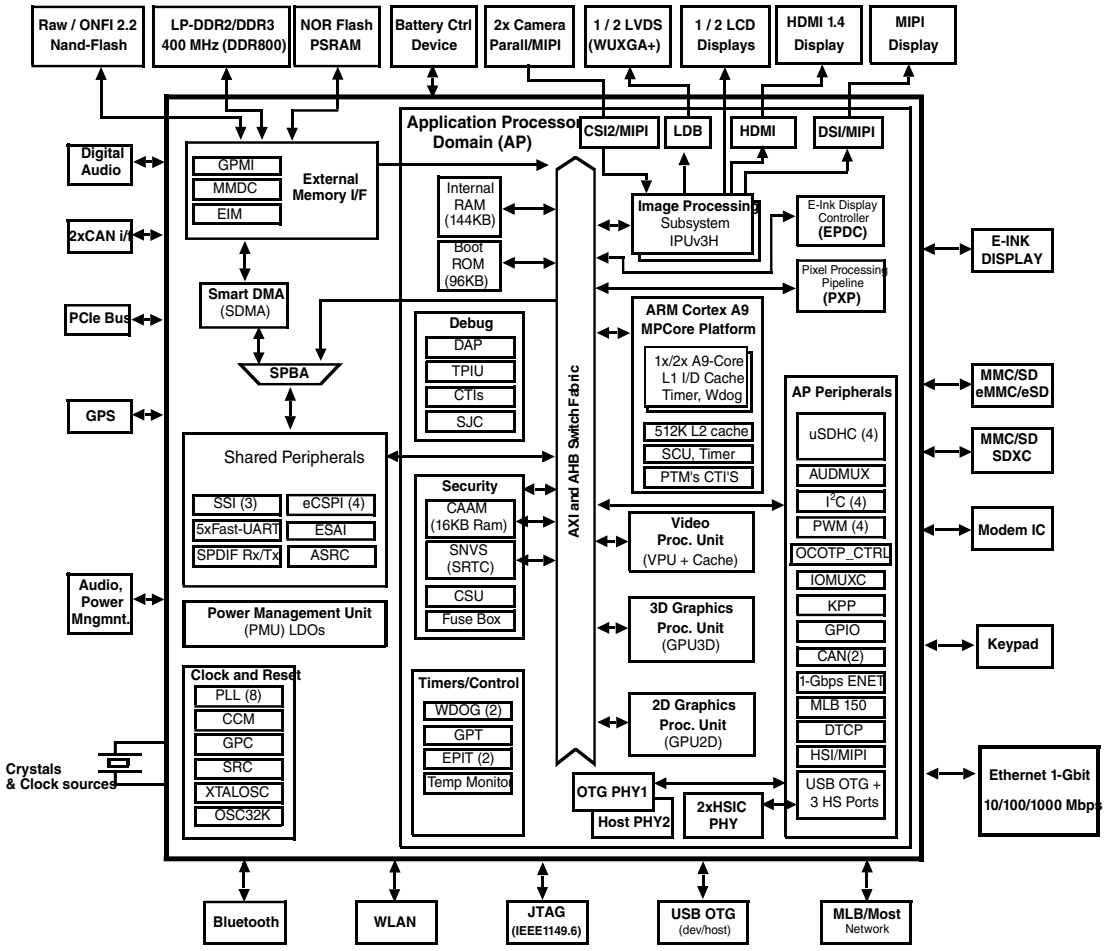


Figure 3.2: i.MX6 DualLite SoC Overview

Despite not possessing FPGA-fabric that allows for on-the-fly deployment of hardware modules, these boards come equipped with a plethora of hardware accelerators aimed at graphics processing. These are a Video Processing Unit, 2D and 3D Graphics Processing Units (GPUs), an Image Processing Unit, connected to various display peripherals, and a Pixel Processing Pipeline, which is required to support Electronic Paper Display (EPD) applications. Cementing this multimedia-driven architecture, an Enhanced Serial Audio Interface, three I2S/SSI/AC97 interfaces and a Audio Multiplexer also provide a fast solution for processing and transmitting audio. For more external general connectivity options, i.MX6 devices come equipped with three USB

hosts, a Gigabit Ethernet Controller, five UARTs (Universal asynchronous receiver-transmitter) and GPIO with interrupt capabilities.

Security is also enforced through a selection of hardware modules. The High Assurance Boot (HAB) ensures the device boots safely, preventing the execution of unauthorized software during the boot sequence and initializing the system accordingly. Masters' access to peripherals is regulated by the Central Security Unit (CSU). Distinguishing security attributes as Secure/Normal and privilege as Supervisor/User, this unit allows developers to assign each peripheral to one of the four possible types of accesses—Secure-Supervisor, Secure-User, Normal-Supervisor and Normal-User. These configurations can be either be set and locked in the CSU registers during the boot process or changed at runtime. As for securing memory, both the 256KB of OCM and external 1GB of DRAM rely on TrustZone protection, through the OCM controller and a TZASC, respectively.

This is the second of the evaluated platforms since it has the same Armv7-A 32-bit Cortex-A9 MPCore processing unit that ZYBO implements, which results in a more comparable version of LTZVisor (unlike an Armv8-A 64-bit implementation), while implementing a different TrustZone approach to the memory subsystem.

3.1.3 ZCU102 Evaluation Board

The Xilinx Zynq Ultrascale+ ZCU102 evaluation board is based on the Zynq Ultrascale+ MPSoC (multiprocessor system-on-chip) architecture, more specifically the EG device family of MPSoCs (Xilinx, 2018). Building upon the APSoC architecture, that integrates software, hardware and I/O programmability through its main subsystems, the PS and PL, the Ultrascale+ MPSoC architecture represents Xilinxs' response to the exponential growth in performance requirements, both in terms of processing power and data transfer rates (Xilinx, 2014). These devices are designed to improve current solutions in fields such as wireless communication that rely on more than one physical platform, integrating them all into the same hardware device, allowing for faster processing with lower power consumption.

Processing System-wise, EG MPSoC devices are comprised of a quad-core Arm Cortex-A53 64-bit APU, a dual-core Arm Cortex-R5 32-bit real-time processing unit (RPU) and an Arm Mali-400MP2 graphics processing unit (GPU). These three main subsystems in the PS side allow for high performance multi-processing, enabling the deployment of complex real-time applications, along with advanced graphics processing. To complement this approach, the PS is divided in three encapsulated power domains (Xilinx, 2018, Figure 3.3)—the Battery Power Domain (BPD), the Low Power Domain (LPD) and the Full Power Domain (FPD)— which can be configured and controlled through the Platform Management Unit (PMU). The BPD is the lowest power domain and it only comprises the Real-Time Clock and Battery-Backed RAM. The LPD, which contains the RPU, a low-power DMA, the PMU and the Configuration Security Unit (CSU), along with Low-Speed I/O and Static Memory Interfaces, is the second domain in the ascending power consumption scale. The FPD is the highest power domain and has access to components such as the APU, GPU, a full-power DMA, the Dynamic Memory Controller and High-Speed I/O, on-top of the LPD and BPD components.

Similarly to the 32-bit dual-core Cortex-A9 processor present in the previous two boards, the 64-bit quad-core Arm Cortex-A53 also provides each of its cores their own 32KB L1 instruction and data cache, with an additional 1MB of unified L2 cache shared among them. Based on an Armv8-A processor, it is the only of the three main processing units with TrustZone support, which could be a problem but, since both the Arm Cortex-R5 and the Arm Mali-400MP2 GPU implement an AMBA3 AXI bus interface (ARM, 2011a, 2018a), they can directly be placed in this TrustZone-enabled SoC design without additional logic—they will not drive the AXI AxPROT[1] signal but maintain the value as it arrived to them, making it possible for propagation across the entire system.

Following the increase of PS resources in Ultrascale devices, the PL also houses higher-density and high-performance banks, almost tripling the amount of maximum logic cells when compared to Zynq-7000 devices (Xilinx, 2016b), power-efficient transceivers and DSP blocks. As with Zynq-7000 devices, the PS and the PL reside on separate power domains, making it possible to power down the PL for power management purposes. The Ultrascale architecture also

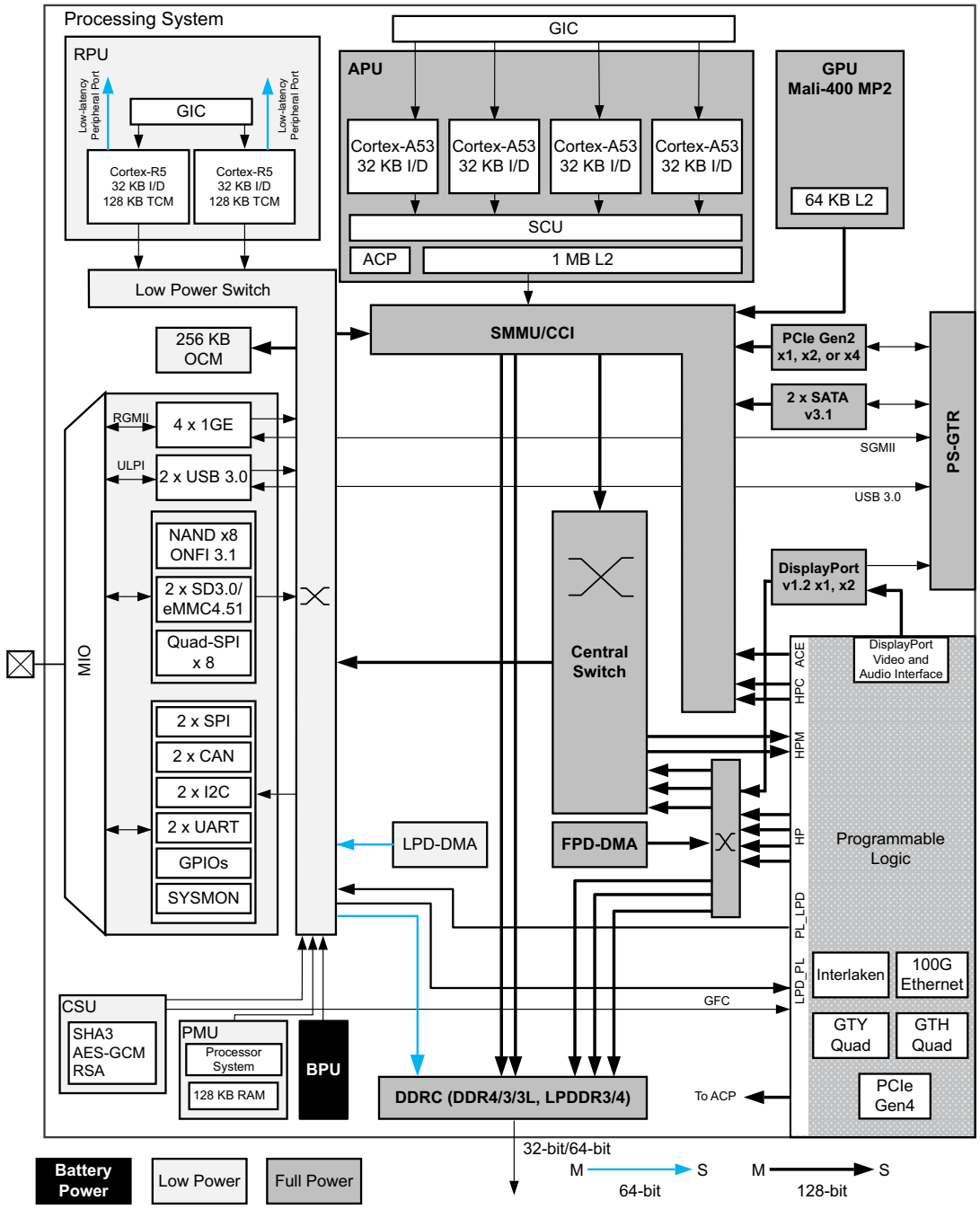


Figure 3.3: Zynq Ultrascale+ MPSoC Block Diagram

introduces high-performance peripheral interfaces in the PL subsystem that allow for transmission rates of hundreds of gigabits per second, such as 100G Ethernet or the Interlaken.

Regarding security, Ultrascale devices mainly implement two types of protection units to safeguard peripherals and memory: the Xilinx Peripheral Protection Unit (XPPU) and the Xilinx Memory Protection Unit (XMPU). Both these units are capable of enforcing security permissions (i.e.

TrustZone secure/non-secure attributes) and masters' privileges, denying transactions that do not meet the security policy or from unauthorized masters.

Comparatively to the two previous boards, this architecture presents a much higher degree of complexity, when viewed in an holistic panorama. To maintain the focus of this work, further discussion of this platform will be approached from the view point of the APU, to establish common ground with the evaluation of the previous two boards.

3.1.4 RaspberryPi 3B

RaspberryPi 3B, released in 2016, is the third-generation RaspberryPi board and the first one to support the Armv8-A architecture. This single-board computer is a low cost platform that is used worldwide by tech enthusiasts who seek a small but powerful device with media connectivity options. Ranging from gaming consoles, surveillance systems, control stations for climate systems, home router, digital picture frames and even pet feeders, it really is a versatile system that can be the central processing hub for various applications.

The RaspberryPi 3B hosts a high-performance quad-core Arm Cortex-A53 cluster that is complemented with 4GB of RAM, on-board Wi-fi, Bluetooth 4.2 and Ethernet, four USB ports, HDMI and a 40-pin GPIO. Based on the Armv8-A architecture, this processing unit implements the Arm security extensions, reason why this platform was a possible target of this study but, outside of the processor domain, there is no further implementation of TrustZone extensions in the memory subsystem. As such, the inclusion of RaspberyPi 3B in this work serves as the example of boards which, although possessing a TrustZone-enabled processor, do not implement a system wide separation between secure and normal worlds. Still, TrustZone applications that solely rely on the extensions present in the APU (MMU/caches) can still be deployed in such devices or even in a more educational context to familiarize a developer with the inner workings in developing TrustZone-aware software.

3.2 LTZVisor

LTZVisor is a Lightweight TrustZone-assisted Hypervisor that relies on the TrustZone hardware extensions as means to implement a virtualized environment (Pinto et al., 2017b). Developed for Armv7-A architecture—currently being ported to Armv8-A and Armv8-M—and supporting execution in Zynq-7000 devices— support for Ultrascale+ and i.MX6 processors also currently deployed but not publicly available—, it demonstrates how TrustZone can adequately be exploited as a virtualization tool that meets real-time needs without hindering performance of hosted rich operating systems.

3.2.1 Architectural Overview

By leveraging on the physical separation between secure and normal worlds in TrustZone architecture, LTZVisor aims to implement three fundamental principles:

- The principle of minimal implementation: Minimizing the trusted computing base of the system by offloading every possible feature to hardware, thus reducing the attack surface for hackers trying to exploit code vulnerabilities.

- The principle of least privilege: Components must be given access only to strictly required resources (e.g., I/O devices, system services, etc), promoting privileged execution and hardware-enforced isolation of the real-time environment from the non-real-time one.

- The principle of asymmetric scheduling: Ensure that timing constraints are met, through the implementation of an asymmetric scheduling policy, where the secure environment has a higher privilege of execution than the non-secure one.

Complying with these guidelines, LTZVisor's architecture is split into three main software components: the hypervisor, the secure VM and the non-secure VM. Privileged software (hypervisor and secure VM) runs in the secure world while the normal world hosts non-privileged software (non-secure VM)— as presented in (Pinto et al., 2017b, Figure 3.4).

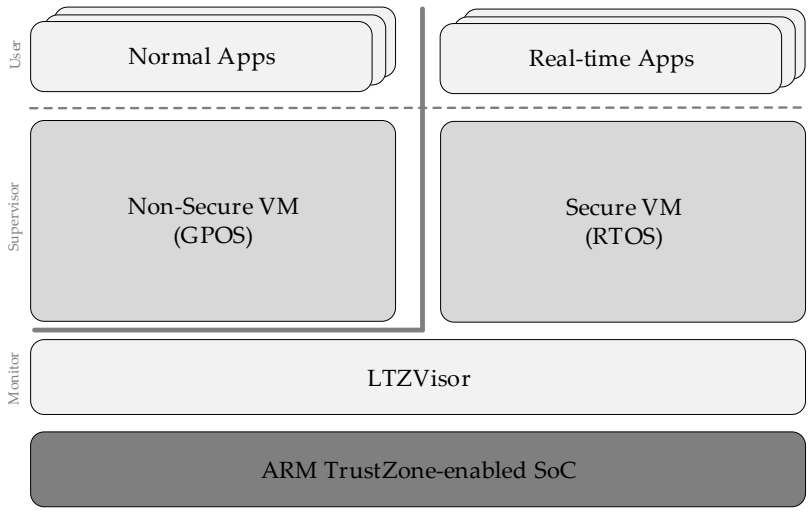


Figure 3.4: LTZVisor Architecture

As LTZVisor may control/regulate access to every single component, it is logical that it runs in the most privileged processor mode, monitor mode. This gives the hypervisor the power to access all hardware or software resources, secure or non-secure, which it requires for setting up the memory, interrupts and devices associated to each VM, as well as managing the Virtual Machine Control Block (VMCB) at every context switch. Whenever the processor is about to run a virtual machine, the hypervisor must restore that VM state to the physical processor context and, likewise, save the current active VM state in the respective VMCB.

Also running in the secure world, but in a less privileged level—the supervisor mode—, the secure VM needs to be aware of the virtualization environment it is placed in. This because while running in the secure world, the VM has access to the non-secure memory or to memory-mapped devices and, as such, can modify them. This fact makes the OS hosted on the secure VM part of the system's Trust Computing Base TCB, slightly increasing its footprint (Pinto et al., 2017b). These make the secure VM ideal to run an RTOS, since the higher privilege of execution—as per the principle of asymmetric scheduling—attributed to the secure world helps meeting its timing requirements. Given that RTOSes usually have rather small memory footprints, it ends up being a win-win situation for both the hypervisor and the RTOS.

Lastly, the non-secure VM, also running in supervisor mode but in the normal world, is architecturally fit to accommodate a more general purpose guest OS. The isolation inherent to running in the normal world means that there is no risk of interfering with another VM's resources. If the non-secure VM ever attempts to access secure memory or peripherals, an exception is immediately triggered to the hypervisor, which assumes control and deals with the situation accordingly.

3.2.2 Implementation Overview

To have an general comprehensive idea of how LTZVisor works, one mainly needs to understand five concepts and how they are implemented: the concept of Virtual CPU in a TrustZone environment; the scheduling policy; how memory is partitioned and managed between both worlds; partition of devices and management of interrupts.

3.2.2.1 Virtual CPU

As discussed in sections 2.2.1 and 2.3, TrustZone's separation of worlds at processor level is implemented through the use of banked registers that are assigned to each world, meaning that the processor only sees the copy assigned to the current (security) state of execution. As such, only the registers which are shared between normal and secure worlds need to be saved in each VM's control block (VMCB) for context switching purposes. These include the General Purpose Registers (R0-R12) and the Stack Pointer (SP), Linker Register (LR) and Saved Program Status Register (SPSR) of each execution mode.

With the purpose to reduce interrupt latency from the RTOS side when resuming execution, there is an important design choice implemented in LTZVisor which differentiates the secure and non-secure VMCBs: even though the non-secure VMCB contains all of the mentioned registers, the secure VMCB solely saves the general purpose registers and one SP, LR and SPSR, allowing for a faster context switch between VMs.

3.2.2.2 Scheduling

LTZVisor's scheduling policy guarantees that the non-secure guest OS is only scheduled during the idle periods of the secure guest OS, and that the secure guest OS can preempt the execution of the non-secure one. In fact, scheduling is controlled by the secure VM and not the LTZVisor itself, which doesn't take away LTZVisor's execution privilege but configures it more as a passive monitor which is responsible for dealing with context switches.

When the secure guest halts execution, it issues a Secure Monitor Call (SMC) to the hypervisor signalling that the non-secure guest can now run until the secure timer tick interrupt configured by the secure guest is triggered. This is possible because of the extra interrupt capabilities implemented in a TrustZone-enabled platform as in the Interrupt Management section below.

3.2.2.3 Interrupt Management

The General Interrupt Controller (GIC) is extended in TrustZone-enabled SoCs to allow for the configuration of secure and non-secure interrupts. Besides the GIC also allowing secure interrupts to have higher priority than non-secure ones, LTZVisor still configures interrupts from secure devices as FIQs and from non-secure devices as IRQs, to get the lowest latency possible from the RTOS side.

When a secure interrupt is triggered while the RTOS is running, it is directly handled, without the hypervisor intervening. When a secure interrupt is triggered while the non-secure GPOS is running, execution is halted and redirected to the hypervisor, which will handle it. Non-secure IRQs will always be handled by the non-secure guest whenever it is schedule to run, regardless of which VM was running when the trigger happened.

3.2.2.4 Memory Partitioning

Since TrustZone-enabled platforms that do not support Arm's virtualization extensions only provide MMU support for single-level address translation, memory partitioning is guaranteed by

a TZASC-like component. This ensures the normal world does not access secure memory. Unfortunately, as previously mentioned, the secure VM has to know the memory boundaries so that it does not tamper with non-secure memory.

LTZVisor does not designate strict memory boundaries since different platforms have different resources and constraints that may or may not allow for the same partitioning initially implemented for the Zynq-700 devices. This is further discussed in the fourth chapter.

3.2.2.5 Device Partitioning

In a TrustZone environment devices can be configured as secure or non-secure, in a static or dynamic configuration, depending on the SoC design. To ensure strong isolation between them, LTZVisor does not allow guest partitions to share devices. As such, devices are assigned to each partition at design time and configured in the boot process: devices used by the secure VM are configured as secure while devices assigned to the non-secure VM are configured as non-secure.

3.2.3 Modifications to the LTZVisor

Besides the changes present in each ported version of the LTZVisor to the three different boards, some modifications had to be carried out in order to evaluate the target architectures. This section focuses not on the differences in the three LTZVisor versions that were deployed in each board but on the necessary adaptations to run the proposed experiments. Thus, the main modifications relate to the abort handling scheme, the enabling of MMU and caches and to the creation of non-secure guests binary images.

3.2.3.1 Abort Handling

Since the errors detected by the TrustZone protection units trigger an abort exception in the processor, it is necessary that these aborts do not totally halt system execution if the dynamic run-time interactions are being evaluated. Given this situation, it was necessary to change the existing basic abort handlers which stopped all execution, trapping it in a continuous loop. This behaviour can be identified by looking at the monitor exception vector table (introduced in section

2.2.1.4), since the default LTZVisor behaviour is to route aborts to monitor mode, whose exception table is represented in the listing below, taken from the assembly implementation of the monitor software.

Listing 3.1: LTZVisor Monitor Exception Vector Table

```
1 _monitor_vector_table:
2   b     .                         @ Not available on MON
3   b     .                         @ Not available on MON
4   ldr   pc, __mon_smc_handler
5   ldr   pc, __mon_prefetch_handler
6   ldr   pc, __mon_abort_handler
7   b     .                         @ Reserved for HYP
8   ldr   pc, __mon_irq_handler     @ Should never come here
9   ldr   pc, __mon_fiq_handler
```

As pictured above, the original LTZVisor implementation prompts a jump to a defined address each time an abort is received by the processor. This allows the usage of an external function that can be defined in another file as the handler function, which in this case corresponds to a C function that prints descriptive text and locks the execution in an endless cycle. This behaviour could also be obtained with the jump instruction "b .", seen in listing 3.1 for the reserved/unavailable entries, that can also trap execution to the current address, but without providing information of the origins of the abort. A more comprehensive external handler can be seen in listing 3.2, where a set of external functions that read the values written to the data fault status registers that provide information about the origin of the aborts. These values are useful from a debugging point-of-view or in an implementation with a health monitoring service, where they are used by the monitoring software to choose the appropriate measures to maintain system stability.

Listing 3.2: Example Abort Handler written in C

```
1 void mon_external_abt_handler(void){
2   printk("\t -> LTZVisor: External Data Abort Exception\n");
3   dfsr_dfar_interpreter(); //checks Data Fault Status Registers
       for the origin of the abort
4   spsr_interpreter();  //verifies the Status Program Status
       Register at the time of the abort
5   while(1);
```

```
6 }
```

As one of the objectives of this thesis is to test the interactions of dynamically changing the TrustZone protection parameters at run-time, stopping the execution every time an error is triggered is not always the desired behaviour. As such, the intended approach was to disable aborts from triggering an exception in the processor by masking them using the CPSR.A bit. According to the Arm documentation, if this bit is set aborts are never taken until the bit is cleared. Unfortunately, when testing this behaviour, it was only true when the MMU was enabled, which would reduce the testable cases. The workaround to this policy was to route execution to the instruction following the one that triggered the abort. To do so, the ldr instruction in line 6 on listing 3.1 was changed to a SUBS instruction that returns to an instruction relative to the address saved in the linker register when the exception was triggered. As introduced in section 2.2.1.2, when a mode change is triggered (in this case, aborts are taken to monitor mode), the program counter is stored in the LR register of the target mode, as well as the current program status register and stack pointer, which the SUBS instruction also restores when returning to the original mode. In the case of aborts, the saved address is two instructions beyond the one that triggered the exception, which corresponds to two words, or eight bytes. So, the instruction placed in the vector table is SUBS PC, LR, #4, which routes the execution to the instruction immediately after the one that caused the abort.

3.2.3.2 Enabling Virtual Address Translation

The base version of LTZVisor runs with the MMU disabled to ensure a deterministic and real-time execution environment. This restriction is imposed in the secure side, while non-secure guests are free to enable and use a virtual memory translation regimen. When using the MMU to perform address translations, it is essential that the devices' memory layout described by the page tables corresponds to the actual memory hardware limitations, in order to avoid system failure. An example of how this could affect system behaviour would be to describe addresses mapped to device access as cacheable addresses. As such, even though the process to create the page tables can be the same for all boards, the layout will differ for all of them, since the

address mappings of devices, ROM, OCM and DDR differ between the boards and so does the mapping of guests for each LTZVisor version.

An approach that was followed to describe the memory layout implemented for the i.MX6 board can bee seen in the listings below, where each entry describes a 1MB section in a one-to-one mapping, i.e., the initial and final addresses are the same, only differing in access permissions or cacheability. These access parameters are set by bits 19-0, since matching a 1MB section only needs the first twelve bits, and their encoding can be saved in macros to ease the implementation, as represented in listing 3.3, where the fields such as IS_A_SECTION or S_NS are bit masks to the corresponding positions in the descriptors for describing a 1MB section or a non-secure section respectively.

Listing 3.3: Page Table Descriptors Assembly Macros

```
1 .set S_KERNEL1, (S_AP1 |S_AP0 | IS_A_SECTION | S_TEX2 | S_TEX0 |
      S_B) //write-back
2 .set S_KERNEL2, (S_AP1 |S_AP0 | IS_A_SECTION | S_TEX2 | S_TEX1 |
      S_C) //write-through
3 .set NS_KERNEL1, (S_NS | S_AP1 |S_AP0 | IS_A_SECTION | S_TEX2 |
      S_TEX0 | S_B)
4 .set NS_KERNEL2, (S_NS | S_AP1 |S_AP0 | IS_A_SECTION | S_TEX2 |
      S_TEX1 | S_C)
5 .set S_STRONGLY_ORDERED, (S_AP1 | S_AP0 | IS_A_SECTION)
6 ...
```

Then, since only the bits 31-20 are used to identify the section, a previously initialized counter can be used to increment the addresses and ensure a one-to-one mapping by shifting its value 20 bits to the left. In the LTZvisor version deployed in this board, the secure guest can occupy addresses between 0x1000_0000 and 0x3000_0000, part of the DRAM, while the non-secure guest would populate addresses 0x3000_0000 and above, as represented on listing 3.4. In this example, both the sections are defined as write-back cacheable. The non-cacheable sections refer to addresses mapped to the ROM and peripherals, which in these case are both defined as strongly ordered memory.

Listing 3.4: Page Table Definition for i.MX6 Board

```
1 ...
2 .section .mmu_tbl,"a"
3 .globl  s_main_page_table
4 s_main_page_table:
5
6   .rept (0x9) //ROM & PERIPHERALS -> 0x0000_0000 to 0x0090_0000
7     .word (count << 20) | S_STRONGLY_ORDERED
8     .set count, (count + 1)
9   .endr
10
11  .set count, 0x9 //OCRAM -> 0x0090_0000 to 0x00A0_0000
12  .word (count << 20) | S_KERNEL2
13
14  .set count, 0xA //PERIPHERALS -> 0x00A0_0000 to 0x1000_0000
15  .rept (0xF6)
16    .word (count << 20) | S_STRONGLY_ORDERED
17    .set count, (count + 1)
18  .endr
19
20  .set count, 0x100 //S_DRAM -> 0x1000_0000 to 0x3000_0000
21  .rept (0x200)
22    .word (count << 20) | S_KERNEL2
23    .set count, (count + 1)
24  .endr
25
26  .set count, 0x300 //NS_DRAM -> 0x3000_0000 to 0x4000_0000
27  .rept (0x100)
28    .word (count << 20) | NS_KERNEL2
29    .set count, (count16 + 1)
30  .endr
31 ...
```

To enable the MMU, the address given by the s_main_page_table label must be used to initialize the translation table base address (TTBR) before using the CP15 registers to activate the MMU and the caches. If this value is not properly initialized, the software will not know where to search for the address descriptors and will not be able to perform the translations.

3.2.3.3 Guest OSes

Given that the LTZVisor provides only the hypervisor and a modifiable secure guest implementation, a non-secure guest must be built externally and then imported by the LTZVisor at

compile-time. To do so, LTZVisor expects a binary image to be placed on a defined location so that the hypervisor knows where to route execution on a context switch. As for the secure guest, it can be also be built externally or with the hypervisor.

In order to avoid conflicts with the LTZVisor implementation of utilities, the guests were deployed using the same libraries and toolchains as the hypervisor and, given that the purpose of this thesis is to test the interactions between the guests in both worlds, these were kept as rather simple bare-metal implementations that allow for a clearer debugging process. Adding a RTOS or a GPOS would not be advantageous, given that the properties being evaluated do not rest on the complexity of the guests but rather on the architectural choices of the TrustZone and the SoC implementations.

As such, for each behaviour that was being tested there were adaptations to the secure and non-secure guest implementations but, in an holistic view, the guests can be reduced to software issuing secure and non-secure memory accesses to pre-defined locations. Thus, by combining different configurations, such as enabling cache on only one world at a time or using different page descriptors in each guest, the implications of the evaluated properties could be compared and evaluated.

4. TrustZone Memory Subsystem: Main Memory

This chapter focuses on the implementation of TrustZone extensions in the main memory subsystem, below cache level. The TrustZone-enabled IP developed by Arm to secure these larger memory blocks is reviewed, according to the documentation provided by Arm, before three different implementations in TrustZone-aware platforms are presented and compared.

4.1 TrustZone-enabled IP

The perception of TrustZone's security state relies on the introduction of a 33rd address bit that can be propagated all throughout the system. This is accomplished by turning the AXI bus fabric that connects the various system components into a TrustZone-aware interface. That is the purpose of the AWPROT[1] and ARPROT[1] signals, for write and read transactions respectively—a low-driven signal means a transaction is secure while a high matches to a non-secure access. As such, the implementation of TrustZone-enabled IP depends on its ability to identify the state of the AxPROT[1] signals, which effectively means it must at least possess a slave AXI interface.

The two main TrustZone-enabled IP blocks that Arm provides for securing memory, the TrustZone Memory Adapter (TZMA) and the TrustZone Address Space Controller (TZASC), function in a region-based approach, allowing the partitioning of memory into secure and non-secure segments, but with very different characteristics in terms of number of regions, granularity and configurability. Both of these controllers receive external signals to configure the region security partitioning, which is then enforced through comparison between the configured address ranges and the issued address.

The TrustZone Memory Adapter was developed with the purpose of allowing a flexible partitioning of on-chip memory into a secure and a non-secure region, in order to diminish the constraints that partitioning made at SoC design time presents to the deployment of system software (ARM, 2006). The TZMA allows a memory block of up to 2MB to be sectioned into two different regions—a lower secure region and a higher non-secure region—, according to the secure region size value given by the R0SIZE input, with a minimum size and at incremental steps of 4KB (Figure 4.1).

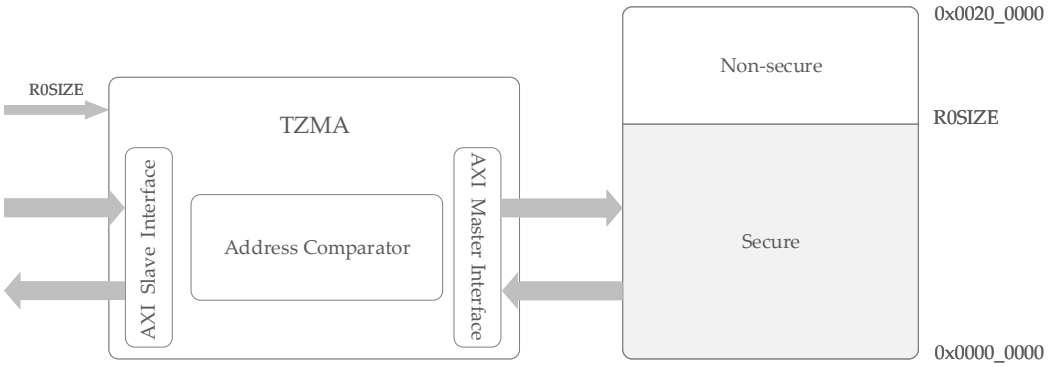


Figure 4.1: TrustZone Memory Adapter

As evident, even though the TZMA is useful in some specific situations, this device is not suitable for larger and more flexible applications, reason why Arm introduced the TrustZone Address Space Controller. The TZASC follows the same principle of the TZMA, meaning that it can dynamically receive a set of configuration parameters to enforce specific AXI access policies, but is fit for a larger and more flexible usage. Whereas the TZMA could only divide a memory block of up to 2MB into two regions, the TZASC supports a variable maximum address range (that should be set accordingly with the system bus width) and a configurable number of regions, where each region is programmable in terms of size, base address, priority and security parameters. Providing up to sixteen overlappable (priority-based) regions with a minimum size of 32KB, that can each be divided into eight subregions, the TZASC can enforce security policies in fine-grained 4KB steps (ARM, 2008). Figure 4.2 illustrates an example TZASC configuration where we can see that region 0 covers the entire address range as the lower priority region. This region cannot be disabled and has no subregions, presenting a fallback to the system, in case all the other

Figure 4.2: TrustZone Address Space Controller Regions

regions are disabled. All other regions are configurable in terms of size and subregion disabling in addition to the security permissions. Besides these improvements, the TZASC also introduces a "Security Inversion" operation mode, which can be very useful in terms of system determinism and isolation. When running with security inversion enabled, in addiction to rejecting non-secure access from accessing secure locations, the TZASC also rejects secure accesses from changing or retrieving non-secure memory locations. This effectively enforces secure world software to issue its transactions according to the security state of memory being accessed, eliminating the possibility of an address coexisting in two different states, which can be exploitable as referenced in section 2.4 when CacheKit (Zhang et al., 2016a) was discussed.

In both of the presented IP blocks, the configuration parameters come from non-AXI signals, which means that they have no control over the validity and security of the permissions they enforce. To ensure this validity, these signal sources must be placed in a trusted location at SoC design time, that will only be accessible by secure world software. For this purpose, Arm

introduced the TrustZone Protection Controller (TZPC), a device that has control over the security state of peripherals connected to the AXI-to-APB bridge, such as the TZASC which receives its configurations signals through an APB interface, and that can also drive the R0SIZE parameter. To enforce this root of trust, the AXI-to-APB bridge must be hard coded to only route secure accesses to the TZPC. If the protection controller is guaranteed to only be accessible by secure software, it can then dynamically inform the AXI-to-APB bridge of which peripherals are to be secure-access only or available for normal accesses so that it can route or reject the AXI transactions to those devices. Its important to note that although this is the approach Arm recommends to the securing of the memory subsystem, these choices are still implementation defined, meaning that any manufacturer can choose to implement their own version of protection devices or architectural design, as discussed in the section below.

4.2 SoC Implementations

As previously introduced, Arm does not force manufactures to abide to a single approach or to even use their own system IPs when implementing TrustZone-enabled SoCs. As a result, different manufacturers, or even different boards from the same manufacturer, present a varied set of features and constraints that may be crucial to the development of system software.

This section reviews three implementations of TrustZone-enabled memory subsystems in three different SoCs, through the boards introduced in Section 3. Their memory subsystems are analysed in terms of granularity, access permissions and dynamism of the configurable protections to the on-board and external memory.

4.2.1 Zynq-7000

The first of the evaluated boards, the ZYBO Zynq-7000 board, manufactured by Xilinx, presents a totally custom implementation to the TrustZone memory subsystem. All Zynq-7000 devices come equipped with a dedicated memory mapped TrustZone module which is used to configure the security permissions of all system components. Xilinx does not disclose many details of how

this controller is implemented, only providing information about how to use the controller, with a set of programmable registers that are used to configure the security permissions.

In terms of on-chip memory (OCM), Zynq-7000 devices support TrustZone regions with a granularity of 4KB. In the ZYBO, the full 256KB of on-chip RAM are sectioned into sixty-four pages which can be configured as secure or non-secure. To do so, the TrustZone module provides two 32-bit registers, TZ_OCM_RAM0 and TZ_OCM_RAM1, where each bit describes the security state of an incremental 4KB page—a value of one identifies a non-secure page while a value of zero signals that that 4KB range can only be accessed by a secure transaction.

As for securing DRAM which is externally connected to the board, although the configuration is also made through the TrustZone module, the supported granularity drops to 64MB sized pages. The register control over DRAM security is similar to the OCM implementation, where each bit of the 32-bit TZ_DDR_RAM register can be set to identify a non-secure 64MB slot or cleaned to signal a secure 64MB region. Since the ZYBO solely possesses 512MB of DRAM, only the least significant eight bits of these register are used to set the permissions to the eight possible 64MB regions. Figure 4.3 displays the overall TrustZone configuration architecture implemented in ZYBO, where we can see the contrast between the support for fine-grained OCM and the coarser-grained TrustZone sectioning of external DRAM.

By clumping the TrustZone system configurations into a set of memory mapped registers under the same module, Zynq-7000 devices simplify, by design, the job of a system programmer. Firstly, since the configuration is set solely through the mapped registers and not with the help of a dedicated protection device, a programmer has no responsibility in terms of initialization, which is required when working with standalone controllers. Furthermore, as they only allow these registers to be accessed by secure software, and which can additionally be locked to prevent any software from changing the registers until the next power-on-reset cycle, there are no concerns regarding nefarious software tampering the memory security permissions.

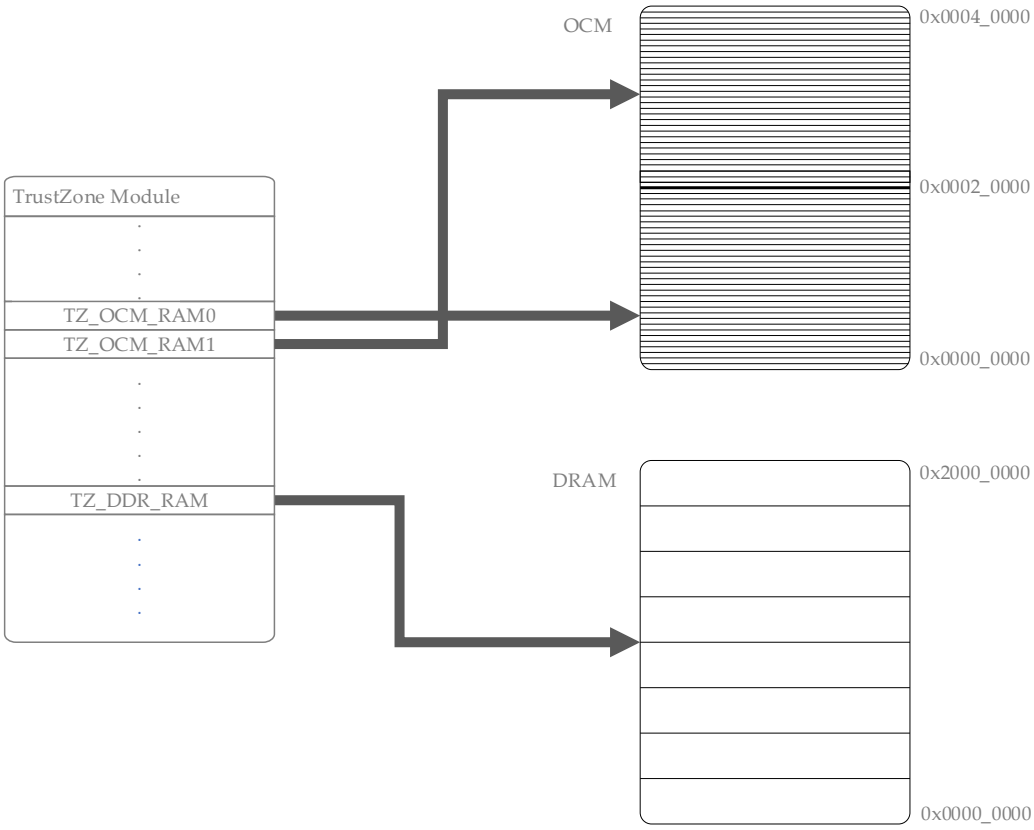


Figure 4.3: ZYBO TrustZone Support for OCM and DDR

4.2.2 i.MX6

Out of the three evaluated boards, the MCIMX6DL-SABRE board, based on the i.MX6 SoC, is the only one that openly uses some of Arm's TrustZone system IP to secure the memory subsystem. In a different approach to the previous implementation in Zynq-7000 devices, that choose to keep all configurations under a single set of memory mapped registers, i.MX6 devices delegate the TrustZone configurations to dedicated controllers that dictate the system's access policies.

Despite not specifying the on-chip memory protection as being handled by a TrustZone Memory Adapter, the approach to securing OCM in these devices is very similar to that of a TZMA. Akin to the TZMA, the OCM controller provides a sizing parameter (with a 4KB granularity) that identifies a secure region within the on-chip memory block but, contrary to the TZMA approach

where the secure region starts on the first address and has a variable size, this implementation provides an input that identifies the starting address of a secure region (OCRAM_TZ_ADDR), extending to the end of the 256KB block (Figure 4.4). This effectively means that while the TZMA separates a block into a low secure region and a high non-secure region, i.MX6 devices support a low non-secure region and a high secure region. The on-chip memory controller also allows the deactivation of this security check through an enable input (OCRAM_TZ_EN) and the locking of both the secure starting address and the enable status until the next power-on reset cycle. These inputs are driven by a set of registers (e.g. IOMUXC_GPR10) under the control of a separate controller, meaning that securing accesses to the TrustZone security configurations of on-chip memory relies on more than the OCM controller, as discussed further below.

Protecting the external DRAM, a TrustZone Address Space Controller instance is connected to each of the two multi mode DDR controllers present in the board (Figure 4.5). These two memory interfaces (MMDC0 and MMDC1) can be used for implementing different memory mapping modes, such as interleaved schemes where the address space is split between two external memory blocks connected to each controller. The two TZASC instances work independently and must each be configured according to the memory mapping mode being employed, meaning that the addresses set in the TZASC configurations must match the ones being issued to the specific controller. As with the on-chip memory approach, the TZASC configurations can also be locked

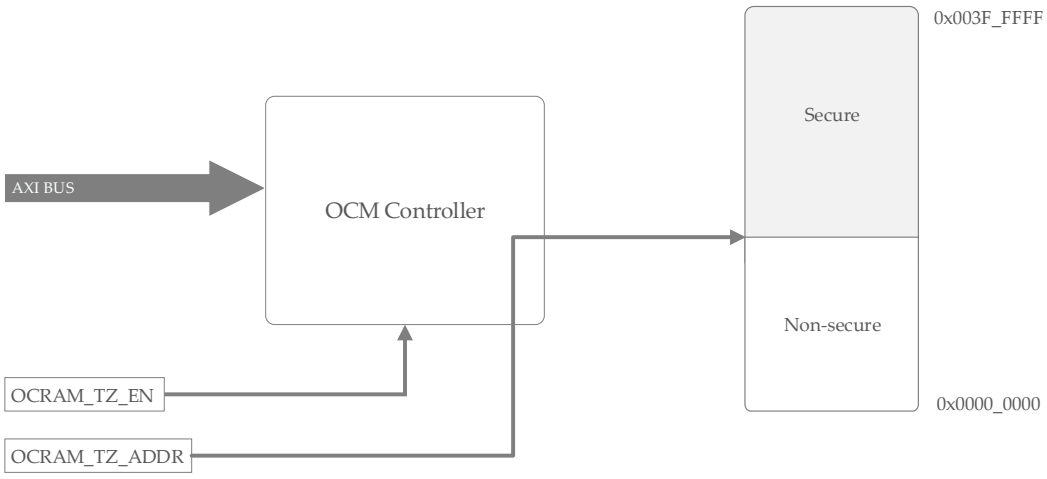


Figure 4.4: i.MX6 OCM TrustZone Support

until the next power on cycle through the respective input and, since this is a dedicated controller, there is also the need to properly configure the clock routing and settings to the TZASC, contrary to the implementations previously presented.

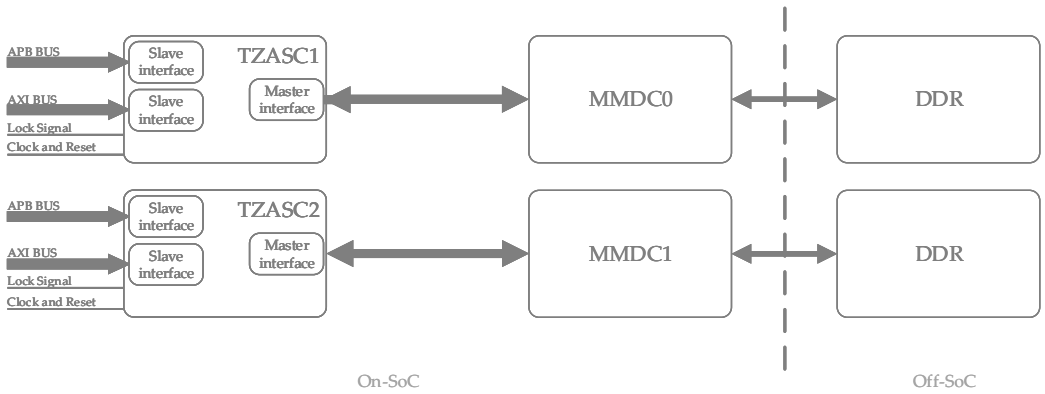


Figure 4.5: i.MX6 DDR TrustZone Support

As introduced in section 3.1.2, i.MX6 devices possess a Central Security Unit (CSU), only accessible by secure software, that dictates the access policies applied to masters accessing system peripherals. This unit acts like a TrustZone Protection Controller and, coupled with a peripheral bridge—alike the configuration previously displayed in Figure 2.14—, sets the security permissions of all devices, including the TZASC and OCM security configurations. This means that, contrary to the approach in the Zynq-7000 devices where modifying TrustZone configurations could only be done by secure software, ensuring that these configurations cannot be modified by untrusted software requires an extra step of identifying and protecting the registers which drive the security input signals to these controllers.

4.2.3 UltraScale+

The third of the surveyed hardware platforms, the ZCU102 evaluation board based on the Xilinx Zynq Ultrascale+ MPSoC architecture, follows an approach to the TrustZone memory subsystem that can be seen as a middle ground between the two previous implementations. This implementation uses dedicated custom controllers to protect system resources from accesses made by specific masters, taking into account their current TrustZone security state. Ultrascale+

devices split the protection of system resources between two types of units, the Xilinx Memory Protection Units (XMPUs) and Peripheral Protection Units (XPPUs), with the memory subsystem protection being under the control of the former.

Xilinx's Memory Protection Units implement a set of region-based memory security mechanisms, including validating the ID of the master accessing memory along with the TrustZone security state of the transaction. Similarly to the TrustZone Address Space Controller, XMPUs can distinguish between sixteen overlappable regions in an address range, configurable as secure or non-secure, following the same priority based scheme of the TZASC. As with the TZASC, these regions can be independently disabled and, akin to the security inversion mode, each region can also prevent secure software from accessing non-secure memory with a secure transaction. Apart from the TrustZone sectioning, each region can also be configured to reject both read or write transactions, regardless of the security state associated with the memory access. Contrary to the TZASC approach, in which a region starting address was specified along with a sizing parameter (minimum 32KB), XMPU region boundaries are set with a starting and an end address, which are restricted by an alignment parameter that may vary according to the characteristics of the protected memory block.

Ultrascale+ boards possess a total of eight XMPU instances protecting the memory subsystem. Six of these are connected to the six available DDR ports (route traffic from masters in the different power domains and programmable logic), while the two other protect the on-chip memory and a set of memory mapped components (Figure 4.6). From the APU viewpoint, three of these instances are used: one for securing the OCM and one securing each of the two the parallel AXI channels connected to the DDR ports. These respectively implement an alignment of 4KB and 1MB, which effectively means that the lowest possible region size is of 4KB for on-chip memory and 1MB for external DDR.

As with the Zynq-7000 implementation previously presented, changing the TrustZone configurations can only be done by secure software despite this more modular approach more in tune with the i.MX6 architecture, which results in a similar level of flexibility but that much more secure

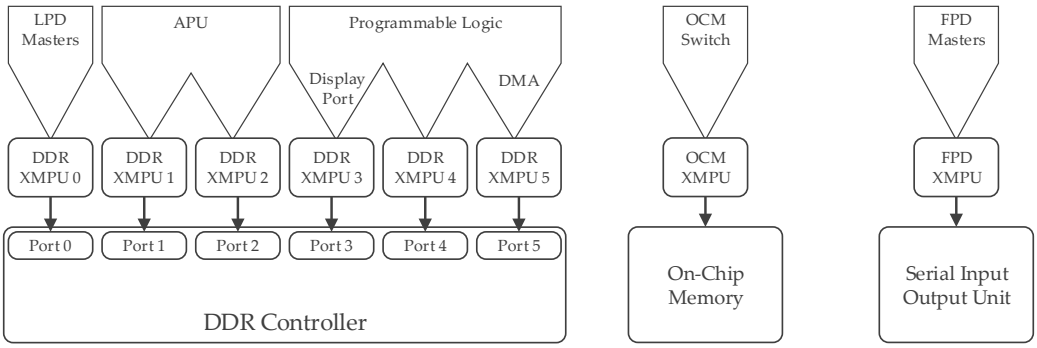


Figure 4.6: XMPUs in the Ultrascale+ Architecture

to implement. Furthermore, a system programmer does not have the concern of clock initialization and routing, programming these units by solely writing to the mapped XMPU registers.

4.3 Discussion

With the review of these three implementations, along with the knowledge that not every TrustZone-enabled platform implements a TrustZone-aware memory subsystem (such as in the RaspberryPi 3B case), the implementation-defined facet of this technology and its inherent porting concerns become more clear. Apart from the different configuration steps required by the various controllers, whose impact can be mitigated by implementing drivers that abstract and standardize the configuration process, the main issues may arise with the disparity in terms of supported granularity, access permissions and dynamism of the target regions. Table 4.1 summarizes the three reviewed implementations by identifying the minimum region size that can be protected for DDR and OCM memory, whether a security inversion mode (only allowing NS accesses to NS memory) can be enabled for regions defined in either of the two memory blocks, if the configurations can be changed by a non-secure access and if the region configurations are modifiable at run time, for each of the evaluated SoCs.

These features were selected as the comparing point between implementations not only because they are the ones which are common to almost all implementations, and whose parameters more clearly vary between them, but also for being the characteristics that present biggest constraints to the development of solutions based on the technology. The following subsections

Table 4.1: SoC Implementations Summary

SoC	Zynq-7000	i.MX6	Ultrascale+
DDR Granularity	64MB	4KB	1MB
OCM Granularity	4KB	4KB	4KB
Security Inversion	None	DDR	OCM and DDR
Access Permissions	Secure-only	Secure and Non-secure	Secure-only
Dynamic Config	Yes (a)	Yes	Yes

(a) only the region security profile is dynamic

describe why and how these different properties can influence the creation, porting and execution of system software to a TrustZone-enabled platform.

4.3.1 Granularity

The granularity of securable regions is one of the parameters that may more obviously present an obstacle to the development of system software and to its porting between different boards. From the TrustZone-assisted virtualization point-of-view we take in this thesis, the biggest impact this attribute imposes is set on the number of possible virtual machines and in the memory fragmentation that the isolation between them creates. The larger the granule size, the less regions can be independently programmed, which in consequence means less VMs can be implemented, relevant in the multi-guest implementations but not so much in classical single/dual-guest approaches. As for the memory fragmentation facet, the larger granule sizes affect all implementations, since having bigger regions simply lowers the percentage of memory usage when compared to regions that more closely match the guest OS sizes. For example, if deploying VMs in a Zynq-7000 device, where the DDR granularity is 64MB, the two worst case scenarios would be implementing a 65MB guest or a 1MB one. In the first case, one guest would be occupying two separate regions, with one of them having 63MB of unallocated memory, which would also happen in the case of a guest size of 1MB.

Another aspect to consider in regards to granularity is the flexibility of region size and starting address. Out of the three reviewed implementations, only the Zynq-7000 had static region placement, where the regions are pre-defined and can only be configured as secure or non-secure—strictly speaking, region 0 of a TZASC also cannot be displaced, but this is a special case

that covers the whole address space as a safeguard. The two other implementations allowed for the overlap of regions and with a flexible address mapping, diminishing the fragmentation problems as in the case mentioned in the previous paragraph. Implementations that allow for a flexible mapping of regions also facilitate the introduction of functionalities to the system, as in the case of Inter-VM communication, which many times even resorts to the usage of shared memory between the guest partitions, besides enabling memory mappings that are well suited to the guest sizes.

4.3.2 Dynamic Configurations

Static versus dynamic configuration regimes always pivot around the same argument: static approaches provide a deterministic aspect that is usually associated with a more secure and higher performance implementation, usually at the cost of functionality, while dynamic systems provide the flexibility necessary to implement more complex solutions but must be more carefully handled so that performance and security metrics are not undermined. This is particularly relevant in the embedded systems realm, where the growing complexity of solutions must still abide to strict timing and security metrics and performance is not measured just in terms of speed of execution.

In the context of this thesis, the dynamism is evaluated in terms of the possible region configurations of security parameters. Every implementation reviewed supports a dynamic configuration of most region properties, with the exception of the Zynq-7000 case, where a programmer can only set the security state of pre-defined regions. The security state of a region is the most important of these properties to be able to dynamically change and is essential when implementing a multi-guest TrustZone-assisted virtualization approach. Changing other properties such as region size and starting addresses at runtime is not as crucial but may still prove to be useful when implementing more complex solutions that rely on flexibility, as in the case of communication procedures between specific VMs. Even so, all the reviewed implementations also provide mechanisms to lock the configurations until the next power-on reset cycle, thus allowing for a

more deterministic implementation in systems that must meet tighter security and performance metrics.

This dynamism does impose some obstacles in terms of coherence that may affect determinism and overall security, mainly when employing a virtual memory architecture that keeps memory locations stored in caches. This is further detailed in chapter 5.2 but its drawbacks can also be observed in a full bare-metal approach where the issued addresses are not translated into their final addresses. Allowing changes to the security properties at runtime may compromise isolation among VMs or even their execution, which could happen if software changed the entire address range to non-secure memory or to secure memory, respectively. As such, it is important that access to configuration registers is made by a trusted source.

4.3.3 Security Inversion

The security inversion characteristic refers to the capability some controllers have of denying transactions marked as secure from accessing memory located in a non-secure region. This does not mean that the secure-world cannot access non-secure memory—as that would be a violation of the TrustZone specifications— but rather that it must issue non-secure transactions when accessing non-secure memory. Out of the three reviewed architectures, only the Ultrascale+ architecture provides this mode to both memory blocks, while the i.MX6 devices allow its usage in protecting the DDR and the Zynq-7000 memory subsystem does not implement such feature in its TrustZone security controller.

This feature may seem useless when in a pure bare-metal environment that does not implement a virtual memory approach with cache support, where a physical address cannot exist in two different states which, as previously mentioned, is not true when working with caches. This is further explored in chapter 5, where this property is described to provide a safeguard to possible misconfiguration of the virtual memory mapping between the guests and the hypervisor that can result in inconsistencies around the current true state of a memory address location. In an approach that does not use virtual memory, this property is useful in ensuring isolation between the secure and non-secure VMs. Even though the hypervisor must be considered a trusted source

of execution, the behaviour of secure VMs should not be taken as guaranteed to not tamper with other system resources. In an ideal implementation, secure software would always be trusted, but the increasing use of open-source software and third party services to implement some general features adds an unknown factor that should be accounted for. By using this feature in a bare-metal implementation, since a secure-VM can only issue secure accesses, it would not be able to access memory tagged as non-secure, thus ensuring a secure VM would not compromise non-secure VM execution.

4.3.4 Access Permissions

Out of the attributes referred in this section, the access permissions (allowing non-secure software to change TrustZone configurations), seem to be the least useful and most dangerous. While the more coarse granularity options may diminish the efficiency of the system, the lack of dynamism reduce its functionality and given that the security inversion serves a purpose of increasing determinism, the access permissions have no visible benefit. Allowing non-secure software to meddle with the security configurations cannot be seen as good policy, reason why both the Xilinx platforms do not authorize it but, as evaluated, is possible in the i.MX6 implementation.

Since many of these controllers have APB interfaces for its configuration registers, which do not distinguish secure and not secure accesses, securing access to their configuration must be done on the AXI-to-APB bridge, either statically or through a TZPC-like controller, as previously explained. Leaving this parameter as a writeable register, for a device that is responsible for setting security configurations, shifts the responsibility of setting the configuration access as secure-only to the system programmer, which only adds a point of attack to the system with no foreseeable benefits.

5. TrustZone Memory Subsystem: MMU/-Caches

This chapter focuses on the changes introduced to the virtual memory and cache models in Arm processors by the TrustZone extensions. The modifications to the architecture are introduced and then reviewed, to better understand how they may influence software development and execution.

5.1 Two Address Spaces

As previously presented, TrustZone-enabled processors add a security bit to every issued address, allowing for the distinction between two address spaces: a secure and a non-secure one. In an implementation that does not enforce a virtual memory translation regime, where all issued addresses match the final target physical address, this bit is solely used by the TrustZone IP presented in the previous chapter to block non-secure software from accessing memory marked as secure. However, when implementing a solution that supports a virtual memory architecture, this bit can be used by secure software to issue non-secure memory accesses, effectively acting as non-secure software. The relevancy of allowing secure software to make non-secure accesses is tied to how TrustZone is extended to the cache memory level, as explained further ahead.

To distinguish between the two address spaces, changes were made to the translation process carried out by the MMU that turns a virtual address issued by the processor into a real physical memory address. As such, the table entries that describe how a virtual address must be translated to its target physical address were extended with a non-secure bit that identifies whether the final address will be issued as secure or non-secure, in the case of an address issued by secure

software. When a transaction is requested by non-secure software, this bit is irrelevant for the translation process, since non-secure software can only issue accesses tagged as non-secure. This behaviour is pictured in Figure 5.1, where two possible page table entries for each translation are displayed, one describing a secure page and another describing a non-secure one.

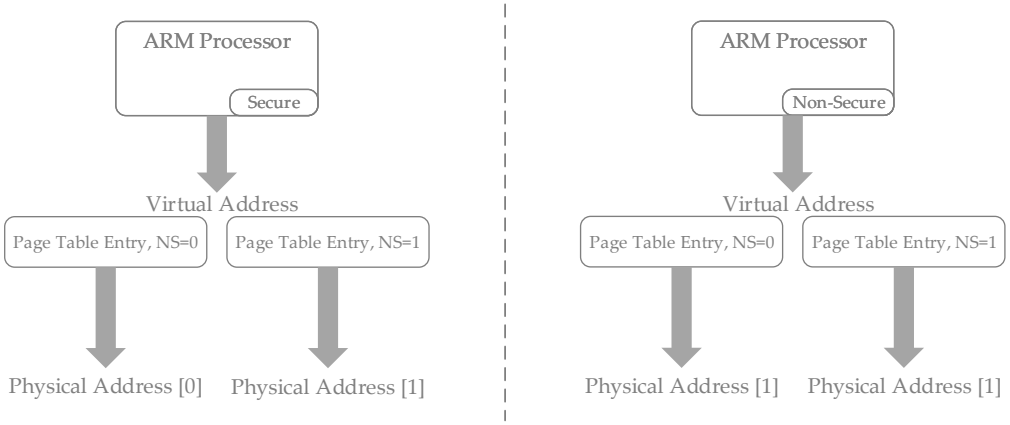


Figure 5.1: TrustZone-aware Address Translation Process

This change in the translation process is also reflected in the Translation Lookaside Buffer (TLB) entries that store recent translations performed by the MMU. With the TrustZone extensions, these units now store the security state of the processor that requested a particular translation, alongside the issued virtual address and its corresponding physical address (extended with the NS bit), as displayed in Figure 5.2. The addition of the world which requested each translation enables TLB maintenance operations that can target specific entries from one of the two worlds. In effect, this reduces the performance drop their execution would pose to the other world, such as when invalidating entries, but also implies more steps when targeting both address spaces, as they need to be executed in both worlds.

TrustZone literature refers to this field as the NSTID (Non-secure Table Identifier), since it tells the TLB which of the translation tables (secure or non-secure copy of the TTBR) was accessed by the processor when performing the translation. This does pose a question on how it can be manipulated and, given the lack of detailed information, it could create some confusion. At first sight, one could assume that this field reflects the state of the SCR.NS bit that dictates the security state of every mode except for monitor mode. If this was the case, since monitor

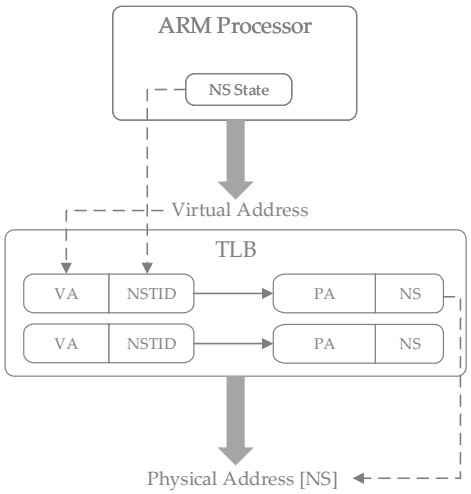


Figure 5.2: TrustZone-aware TLB

mode can technically run with either SCR.NS bit value, it should be able to manipulate this field to operate over either address space, which would ease maintenance targeting both address spaces. This hypothesis was tested by invalidating the TLB at specific points of execution after changing the LTZVisor guest translation tables at run-time and seeing which entries were invalidated, with the results summarized in Table 5.1.

Table 5.1: TLB Maintenance Operations From Different Sources

	Invalidates TLB Entries	
Execution State	**NS Page Table**	**S Page Table**
Guest NS	Yes	No
Guest S	No	Yes
Monitor S	No	Yes
Monitor NS	No	Yes

The results presented above, coupled with also observing that monitor mode always uses the secure copy of the TTBR when translating an address, regardless of the SCR.NS bit status, lead to the conclusion that monitor mode always executes as secure software, even though it has access to the non-secure copies of the banked registers. As such, even though monitor mode can still be used to configure the virtual machine environments prior to their execution, an hypervisor will have to leave monitor mode and switch between security states when performing maintenance operations targeting both address spaces.

5.2 TrustZone-aware Caches

After a translation is performed by the MMU and a final physical address is issued to the memory subsystem, tagged with the corresponding NS bit, it is important to take into account the changes introduced to the cache level, before reaching the main memory. Even though there are some differences between the implementations of level-1 and optional lower level caches, the main modifications are true to all levels and reflect the changes to the address spaces, while the divergences mostly affect the cache maintenance operations.

The architecture of TrustZone-enabled caches is rather simple to understand if we view the same unique physical address as two distinct physical addresses when tagged with different NS fields. In doing so, cache lines will be filled as depicted in Figure 5.3 after receiving a physical address resultant of the translation process. In the figure, the connection between the main memory block and the bus interface is present to represent the implementations that may not include a level-2 cache controller and, as such, would not receive transactions from that source. In any case, these connection points are where a TZASC-like controller would be placed and thus the final destination for TrustZone-aware addresses.

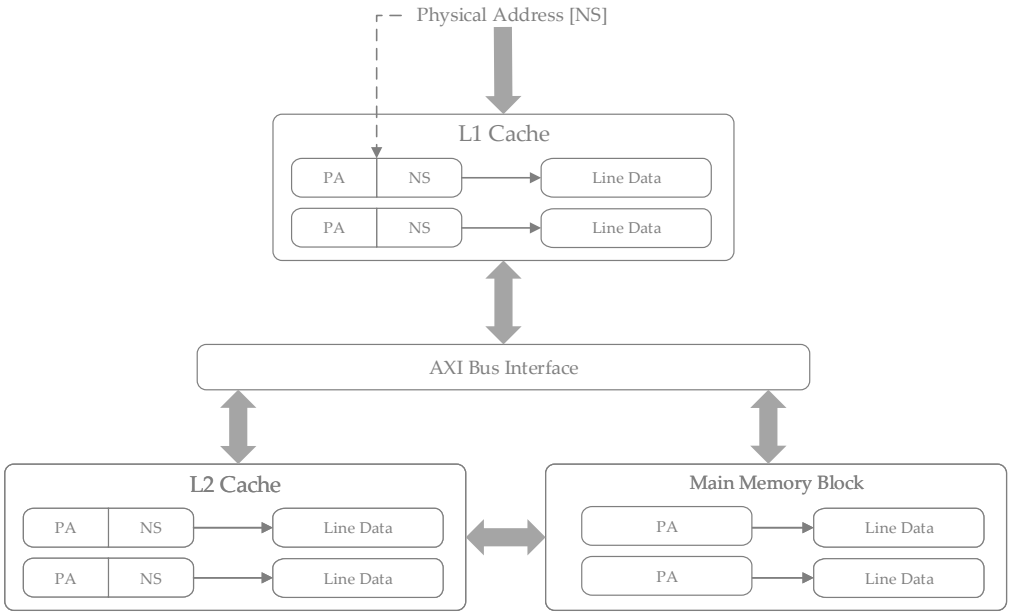


Figure 5.3: TrustZone-aware Caches

According to Arm, adding the feature of allowing the same physical address location to be stored in two different lines, when tagged with different NS bits, was done to increase system performance. If both worlds wished to use caches but these were not extended with the NS tag, there would be a need to flush its contents every time a world switch occurred, in order to guarantee that non-secure software would not access secure information. By tagging every address with the security context of the associated transaction, this need to flush the cache when switching worlds is removed by guaranteeing that non-secure software can only access lines tagged as non-secure, which consequentially increases the system performance. Arm also mentions the increased performance of communication mechanisms between both worlds, which most likely refers to the possibility of both worlds accessing the same non-secure lines (if secure world issues non-secure accesses to a specific region), but this would also be the case if caches were not extended with the NS tag.

Unfortunately, this design choice may present some drawbacks in terms of determinism and security, which are further amplified by the lack of information on how these features are implemented (mainly in L1 caches) since these are hardware implementations to which system software is oblivious. Moreover, this choice of allowing the same address to be described in two different lines drifts away from previous Arm architectural choices at cache level, specifically in the implementation of the write-back and write-through attributes. The implemented policy dictated that it was illegal to map a physical page with multiple cacheability attributes—this would happen when one page table described the address range as write-back and another one as write-though—in order too prevent unpredictable behaviour in the system. As such, no page described as write-through is allocated in the cache but rather directly sent to the AXI master interface. Even so, as detailed below, this mismatch of attributes becomes a concern with the introduction of the TrustZone extensions, as the NS field effectively acts a cacheability attribute that can differently describe a single physical address.

5.2.1 Eviction Policy

One of the first features to raise some concerns in the academic community was the eviction policy of TrustZone-aware caches. As described in official TrustZone documentation, any non-locked down cache line can be evicted to make space for new data, regardless of its security state, which means that it is possible for a secure line load to evict a non-secure line, and for a non-secure line load to evict a secure line. This reflects a choice of not granting a performance privilege to any of the two worlds, which is comprehensible given the fact that preventing non-secure line fills from evicting secure lines could severely impact the non-secure execution, but may present an increase in the vulnerability to cache-based attacks.

As identified in several academic works (Guanciale et al., 2016, Lipp et al., 2016, Zhang et al., 2016b), this eviction policy potentiates cache-based attacks that, through prime-and-probe and timing analysis techniques, are able to retrieve sensitive encrypted information from the secure world. Since two lines describing the same address, one as secure and one as non-secure, can be found in the same set of cache locations, software running in the normal world can monitor the cache movements of the secure world and later compromise the confidentiality or integrity of secure operations. Figure 5.4 illustrates the basic layout of a prime-and-probe attack that can be used to track secure information such as execution/memory access patterns, by monitoring the cache content modifications after secure software runs, in a simple direct mapped cache implementation.

These mismatched attributes can also be exploited from a memory evasion standpoint, where nefarious software is invisible to memory introspection software, as in the CacheKit (Zhang et al., 2016a) example. Relying on the fact that the cache contents of a secure and a non-secure line that are mapped to the same physical address can differ, malicious software can in theory evade checks from the secure world. This banks on assuming that the secure monitoring software only issues secure accesses to memory and, as such, has no view of the non-secure lines, which is easily solved by configuring the monitoring process to issue non-secure accesses.

Workarounds for these cache-based issues usually rely on an hypervisor managing the translation process of its guests, in order to ensure there are no mismatched cacheability attributes.

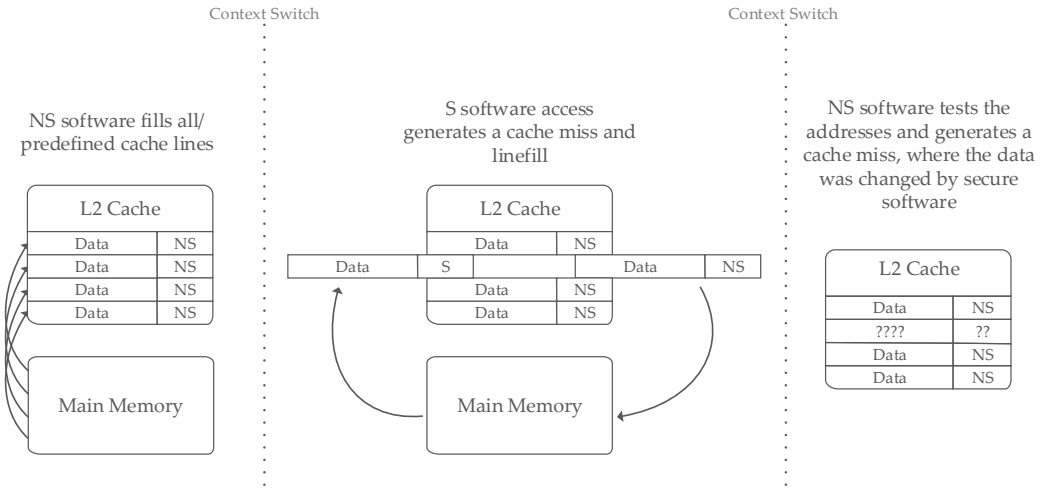


Figure 5.4: Example of a cache-based attack

The most effective way of dealing with this would be to disable caching of the normal world, but, as mentioned, this would come at a very high performance cost for non-secure software. Alternatively, an hypervisor could clear the cache contents before executing trusted code, which also presents a big performance decrease, or otherwise guarantee that all locations accessed by non-secure software are mirrored by the secure-software, i.e., that secure software issues NS accesses to memory locations accessible by non-secure software. The latter would require hypervisor maintenance of both secure and non-secure page tables or an implementation of level-2 translations, so that the non-secure guest has a reduced view of the system restricted to the non-secure memory sections.

A different solution may be implemented in the second cache level with the introduction of locking capabilities that are also TrustZone-aware. As mentioned, locked entries cannot be evicted by line fill requests and, as such, are not susceptible to the mentioned attack types. Level-2 TrustZone-enabled cache controllers can restrict the locking of entries to accesses made by secure software, making those locked secure cache lines impervious to the cache monitoring performed by non-secure software.

5.2.2 Maintenance Operations

The TrustZone extensions add some nuances to the cache maintenance operations, allowing for world specific interactions that take into account the addresses' NS tag. The policy dictates that operations issued by secure software can affect both S and NS lines, while non-secure software can only affect lines tagged as NS.

As with the TLB case, information on the implementation of processor side caches (L1 caches) is scarce and does not answer some important questions around its inner workings, mainly concerning execution in monitor mode and priority when executing secure sided operations. Table 5.2 presents the verified correlation between the affected cache lines and the execution state that performed the maintenance operation. The results obtained are in line with the TLB findings, where monitor mode executes as secure software, regardless of the SCR.NS bit.

Table 5.2: Cache Maintenance Operations From Different Sources

	Affected Lines	
Execution State	**Lines tagged as NS**	**Lines tagged as S**
Guest NS	Yes	No
Guest S	Yes	Yes
Monitor S	Yes	Yes
Monitor NS	Yes	Yes

Level-2 caches, on the other hand, do not present the same results. Since the L2 cache controller is a memory mapped device, executing the maintenance operations is done by writing to a dedicated addressable register. As such, the cache controller checks the security permissions by evaluating the transaction security bit, i.e., if the register's address was issued as a secure or non-secure address. This means that secure software can effectively act as non-secure and issue maintenance operations that only affect non-secure lines, by mapping the register accordingly in its translation tables. This is a very useful feature mainly for the multi-guest virtualization approaches previously mentioned, where the active guest always runs in non-secure mode, and is kept in secure memory locations when inactive. In these implementations, the caches need to be flushed every time a new guest starts execution, to enforce data isolation. If this feature was

not available, every time the hypervisor, running in monitor mode, issued a clean command, it would also clear its own cached addresses, which would negatively impact its own performance.

After answering the questions around monitor mode, the doubt about the eviction priority was tackled. If secure software can clean all cached lines and we know that a secure and non-secure line can be mapped to the same physical address, which of the two lines is written to memory? Does this generate an error, or does TrustZone employ any priority algorithm that ensures the secure line is always the value written to memory? Unfortunately, testing indicates that there is no TrustZone-aware scheme when deciding which line to write in main memory. What this means is that the contents which end up in main memory after flushing the whole cache are only dependant of the current location in the cache and the controller's maintenance policy, i.e., in which order it clears the contents to memory. Since the two lines can be placed in the same sets but in different ways (in a set associative implementation), without making use of cache locking or coloring techniques, where the addresses are knowingly stored in determined cache locations, the final memory value written to memory is never deterministic, being either the secure or the non-secure copy. This further amplifies the relevance of the solutions presented in section 5.2.1 to remove instances of mismatched attributes describing the same physical address.

5.2.3 Interaction with TrustZone-IP

So far this section has analysed the cache subsystem on its own, only mentioning the interaction between the processor and the caches. Even though its relevant to make that analysis, to better understand its nuances and vulnerabilities, and given that not every board implements a fully TrustZone-enabled memory subsystem—as in the RaspberryPi 3B case— it is also important to note that the cache interactions do not end with the L2 controller.

When the memory subsystem is equipped with TrustZone memory protection units, these and the cache work in tandem to prevent some of the situations described in sections 5.2.1 and 5.2.3. Behind this interaction is the abort handling scheme implemented in Arm processors. As previously mentioned, when a non-secure access tries to access secure memory, and external abort is generated to the processor which can then halt execution and handle it the way is deems

best. This is the result of the TrustZone-IP unit which is protecting the memory either triggering an interrupt signal that is connected to the processor or, more commonly, by returning an error (DECERR) in the AXI channel, which is interpreted by the processor as an external abort. When using caches, if the address tag does not match the target region security state set by the TrustZone IP, a line fill will result in no data pulled to the cache and a DECERR response will be sent to the processor, while in the case of an eviction nothing is written to main memory but the line is evicted from the cache and a DECERR response is also issued.

The relevance of mapping non-secure memory as non-secure accessible in secure page tables is further emphasised in the presence of a TrustZone-IP unit that possesses a security inversion mode, but the presence of these units also help mitigate the mismatch between the secure world view and the TrustZone-IP region mapping. In these cases, where secure transactions cannot access non-secure memory, ensuring secure software issues accesses complying with the target region's security state means reducing the number of aborts in the system. Even though it does not strictly cross the architectural rules, TrustZone security aborts triggered by secure software should not exist, as it sort of defeats the purpose behind the existence of two different worlds where secure software is expected to be a reliable execution environment. On the other hand, this security inversion mode prevents the existence of the same physical address on two separate cache lines, by ensuring that a non-secure memory location can only be accessed by a non-secure access, which reduces the vulnerabilities created by this duality approach taken by the TrustZone extensions, as previously described.

5.2.4 Discussion

Running tests to the modifications introduced by the TrustZone extensions to the MMU and cache models by using a dual-guest virtualization approach allows a clear verification of its duality properties. Intrinsic to splitting the address space into two, these modifications are best suited to accommodate two execution environments and, at first sight, the concept of running two OSes atop of the same mirrored address spaces, where they can effectively each make use of the full addressable space in an isolated manner, may seem like a great approach. Unfortunately, this

would only be the case if at least one of the two guests was loaded into the cache and all of its read or write transactions were redirected to the cache, never reaching the main memory subsystem, otherwise creating an indeterministic and unstable environment. This can be accomplished if a guest is small enough to be loaded into the cache, where it can then remain by using available cache locking mechanisms provided by the L2 cache controller.

Given that the presented case is a very specific one, and taking into account that two completely distinct address spaces would hinder communication capabilities between VMs and with external devices, there is a need to make sure the two address spaces can work in tandem with each other. The easiest way to accomplish this is to make sure that non-secure memory is never accessed by a secure transaction. To do so, the configurations set in TrustZone-IP units must be reflected in the page tables used by the secure world, both by OS and monitoring software. As described in this chapter, this approach is the one that mostly reduces the attack surface for cache exploits, while also getting rid of possible TrustZone security aborts generated by secure world software that would also harm performance.

If tweaked with these changes, the approach described in the first paragraph could actually be successful, without hindering communication from the guest locked in cache. Supposing this was the non-secure guest, its communication could be implemented as a system call that would prompt secure software to fetch information from a pre-defined location and then transmitting it to the desired destination, provided that the pre-defined location was described with a non-secure tag in the secure page tables. This way, the secure guest would retrieve data from the non-secure cache lines and distribute it to where it deemed fit. An approach like these can also be employed in boards that do not possess TrustZone-IP securing the main memory subsystem to enforce isolation among VMs without having to deploy more complex page table layouts. Since the non-secure guest would never reach main memory, this could be considered as secure. How feasible or in what cases this might be employed is very dependant of the guest and board specifications, but its still something that might be of value to further analyse.

Unfortunately, when talking about a TrustZone-assisted multi-guest virtualization approach, working with caches is not very efficient, and the workarounds to the challenges are not very

evident, when talking about a system with no hardware support for two-level translation. Running all guests from the normal world while the inactive VMs are kept in secure memory is relatively straight forward, when no caching is involved. But, when the guests are allowed to save data in caches, there is a need to flush the non-secure cache contents every time there is a context switch. Otherwise, guests would have access to each others data, corrupting the isolation between VMs. Without employing a software based two-level translation regime, where the hypervisor would have to ensure each guest only accesses memory allocated to them, this problem persists. In (Martins et al., 2017), the authors show that guest performance can drop more than 20% when compared to their native execution, when using a sub 1ms guest switching rate, due to this need to flush cache contents. This issue is solved in devices that possess hardware support for two-level translation, which is the case for Armv8-A devices or devices with virtualization extensions. In these devices, addresses can be tagged according to the VM that accessed them, preventing VMs from accessing locations to which an hypervisor deems they have no permission to. As such, one could say that TrustZone-assisted multi-guest hypervisors are as viable as single or dual-guest implementations in these boards but this seems sort of unfair, as the extensions that would make these approaches as viable would be the virtualization extensions and not the TrustZone-extensions. This does not mean that multi-guest TrustZone-assisted hypervisors are unusable but rather that they will demonstrate much higher performance degradation when compared with the more classical approaches.

The performance deterioration present in multi-guest TrustZone-assisted virtualization solutions, coupled with the introduction of the Armv8-A architecture, might shift the usage of TrustZone back to a more security-oriented panorama. With the addition of hardware support for two-level address translation to the base architecture, no longer being provided as optable virtualization extensions, and given that register banking between the secure and normal world is also removed in the Armv8-A architecture, the benefits of using TrustZone as the foundation for a virtualization solution no longer seems as prevalent, given the proliferation of Armv8-A devices. As one of the first points made in this document, the impetus to implement hypervisors based on the TrustZone extensions was the large availability of boards with these extensions when compared

with VE-enabled devices. But, following the increasing complexity of software solutions, Arm now provides better ways to implement these more complex systems, such as in the case of multi-guest virtualization, with TrustZone providing the backbone of system-wide security. Although not entirely reliant on this technology, virtualization solutions deployed in Arm devices will still work in tandem with the security extensions to provide reliable execution environments, and will have to abide to the TrustZone policies mentioned in this thesis, at least until the next version of the security extensions.

6. Conclusion

When the Arm TrustZone security extensions were announced, a few researchers saw an opportunity to bring virtualization solutions to a wider range of devices, enabling the deployment of virtualized environments where until then it was not viable to do so. As the interest grew around this technology, more devices were developed with diverse approaches to the implementation-defined TrustZone subsystem. Different manufacturers building different implementations meant that constraints within TrustZone functionalities also varied, and that what was possible in one implementation may not be true in a different one. Along with the general lack of concise information around this technology, grasping its possibilities and limitations can prove to be challenging, with no hub for reliable condensed information.

This thesis attempted to help fill the information gap around this technology, mainly focusing on the TrustZone memory subsystem. Firstly it covered the protection units that are used in different boards for protecting the main memory subsystem—the larger memory blocks connected to the boards through a dedicated controller—,to better portray which properties a system software programmer has available when deploying a TrustZone-assisted virtualization solution, and how these may influence the development or execution of said software. It identified four main properties, in the form of granularity of securable regions, what security profiles can be defined, how dynamically can their security parameters be configured and who has permission to change them. Three board implementations were reviewed, using the LTZVisor as a software platform on top of which tests could be carried out. These showed that there is variety in the implementations, with the Zynq-7000 board clearly prioritizing security and determinism, while the i.MX6 board implemented a very flexible but more fallible approach and the Ultrascale+ board presenting a more balanced implementation.

The second part of this thesis targeted the changes introduced by the TrustZone security extensions to the virtual memory subsystem of MMU and caches, that are crucial for the deployment of a virtual memory system architecture. The division between two separate address spaces was explained, along with the duality introduced to the cache lines that allows the same address to exist in two separate cache lines at the same time. The advantages and drawbacks of these design choices were presented from an isolated and then a more in context point of view, reviewing how they can improve performance but also how they can be exploited to illegally extract information that should be secure. Finally, the problems this architecture presents to the implementation of multi-guest TrustZone-assisted virtualization were discussed, presenting the facts that make this approach less than ideal, given the introduction of the Armv8-A architecture, which helps mitigate the problems created by multiple guests sharing the cache contents.

After ending the last chapter, the notion that TrustZone-assisted virtualization has reached its full potential in the form of single and dual-guest approaches, which take full advantage of the dual world separation introduced by this technology, is ever present. Even so, all of the concepts reviewed remain of critical understanding, as future virtualization solutions based on Arm systems will have to work in tandem with the security extensions in order to provide a robust execution environment, as the concepts of extensions for virtualization and security become once again separate but now in a more intertwined manner.

This document ends up providing a small compilation of TrustZone information, both from reviewing existing documentation and by actually experimenting with the diverse implementations. This will hopefully create a source for anyone trying to understand how a TrustZone-enabled memory subsystem is implemented and what options there are for doing so, as well as presenting some good practices that should be followed when developing software in a TrustZone environment.

On a more personal note, this thesis was a new type of challenge to develop. As a more theoretical work, the way of thinking and planning were quite different from the projects developed so far, as there was a need to figure out not how to reach a certain result but rather how the obtained results can be used to figure out how the TrustZone features are actually implemented.

6.1 Future Work

Extending this work means reviewing the biggest number of different implementations available, in order to widen the discussion around the advantages and drawbacks that each approach presents. This would allow manufacturers to have a larger set of information to draw from when designing new devices, as well as creating a bigger discussion pool so that future improvements to the technology can be applied.

From a virtualization standpoint, the sense is that the Armv8-A technology and its embedded virtualization extensions are the way to keep growing these solutions. As such, it is relevant to study how the findings of chapter 5 translate to a scheme where each cached address can still only exist as secure or non-secure but access to it can be managed for each VM by the translation process. Does this change reduce the attack surface of cache-based attacks or does it only improve performance? In the last few years, caches have been the target of many attempted exploits and study of countermeasures that do not hinder performance seems to be relevant in the research community. This is a topic that, given the increased vulnerability that has been identified in the duality property of TrustZone-aware caches, also seems like a relevant subject to pursue.

References

ARM (2005). ARM Architecture Reference Manual.

ARM (2006). PrimeCell Infrastructure AMBA 3 AXI TrustZone Memory Adapter (BP141). Technical Overview (Revision r0p0).

ARM (2008). CoreLink TrustZone Address Space Controller TZC-380. Technical Reference Manual (Revision r0p1).

ARM (2009a). ARM Security Technology. Building a Secure System using TrustZone Technology. ARM White Paper.

ARM (2009b). ARM Security Technology. Building a Secure System using TrustZone Technology ARM. ARM white paper.

ARM (2011a). Cortex-R5. Technical Reference Manual (Revision r1p2).

ARM (2011b). Technology Preview: The ARMv8 Architecture. ARM White Paper.

ARM (2012). ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition.

ARM (2014). ARM Cortex-A Series. Programmer's Guide (Version 4.0).

ARM (2015). ARM Cortex-A Series. Programmer's Guide for ARMv8-A.

ARM (2017a). ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile.

ARM (2017b). TrustZone – Arm. [Online]. Available: `https://www.arm.com/products/security-on-arm/trustzone` [Accessed: 2017-11-15].

ARM (2018b). Isolation using virtualization in the Secure world. ARM white paper.

ARM (2018c). Q2 2018 Roadshow Slides.

ARM (Accessed on: Dec. 14, 2018a). Graphics and Multimedia Processors: Arm

Mali-400 GPU. [Online]. Available: https://developer.arm.com/products/graphics-and-multimedia/mali-gpus/mali-400-gpu.

Benhani, E. M., Marchand, C., Aubert, A., and Bossuet, L. (2017). On the security evaluation of the arm trustzone extension in a heterogeneous soc. In 2017 30th IEEE International System-on-Chip Conference (SOCC), pages 108–113.

Blem, E., Menon, J., and Sankaralingam, K. (2013). Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on. IEEE.

Douglas, H. (2010). Thin hypervisor-based security architectures for embedded platforms. PhD thesis, Royal Institute of Technology.

EETimes and Embedded.com (2017). 2017 Embedded Markets Study: Integrating IoT and Advanced Technology Designs, Application Development ands Processing Environments.

Frenzel, T., Lackorzynski, A., Warg, A., and Härtig, H. (2010). Arm trustzone as a virtualization technique in embedded systems. In Proceedings of Twelfth Real-Time Linux Workshop, Nairobi, Kenya, pages 29–42.

Goodacre, J. and Sloss, A. N. (2005). Parallelism and the arm instruction set architecture. Computer.

Graziano, C. D. (2011). A performance analysis of xen and kvm hypervisors for hosting the xen worlds project.

Guan, L., Liu, P., Xing, X., Ge, X., Zhang, S., Yu, M., and Jaeger, T. (2017). Trustshadow: Secure execution of unmodified applications with arm trustzone. In Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, pages 488–501. ACM.

Guanciale, R., Nemati, H., Baumann, C., and Dam, M. (2016). Cache storage channels: Alias-driven attacks and verified countermeasures. In 2016 IEEE Symposium on Security and Privacy (SP), pages 38–55. IEEE.

Kauer, B. (2014). Improving System Security Through TCB Reduction. PhD thesis, Technischen Universität Dresden.

Kim, S. W., Lee, C., Jeon, M., Kwon, H. Y., Lee, H. W., and Yoo, C. (2013). Secure device access for automotive software. 2013 International Conference on Connected Vehicles and Expo, ICCVE 2013 - Proceedings, pages 177–181.

Lee, C., Kim, S. W., and Yoo, C. (2016). VADI: GPU Virtualization for an Automotive Platform. IEEE Transactions on Industrial Informatics.

Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. (2016). Armageddon: Cache attacks on mobile devices. In 25th USENIX Security Symposium, pages 549–564.

Liu, R. and Srivastava, M. (2017). Protc: Protecting drone's peripherals through arm trustzone. In Proceedings of the 3rd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications, pages 1–6. ACM.

Lucas, P., Chappuis, K., Paolino, M., Dagieu, N., and Raho, D. (2017). VOSYSmonitor, a low latency monitor layer for mixed-criticality systems on ARMv8-A. Leibniz International Proceedings in Informatics, LIPIcs, 76(6).

Martins, J., Alves, J., Cabral, J., Tavares, A., and Pinto, S. (2017). μRTZVisor: A Secure and Safe Real-Time Hypervisor. Electronics, 6(4):93.

Masmano, M., Ripoll, I., Crespo, A., and Metge, J. (2009). Xtratum: a hypervisor for safety critical embedded systems. In 11th Real-Time Linux Workshop, pages 263–272. Citeseer.

Ngabonziza, B., Martin, D., Bailey, A., Cho, H., and Martin, S. (2016). Trustzone explained: Architectural features and use cases. In Collaboration and Internet Computing (CIC), 2016 IEEE 2nd International Conference on, pages 445–451. IEEE.

NXP (2017). i.MX 6Solo/6DualLite Applications Processor Reference Manual. NXP Semiconductors.

Pinto, S., Araújo, H., Oliveira, D., Martins, J., and Tavares, A. (2019). Virtualization on TrustZone-Enabled Microcontrollers? Voilà!

Pinto, S., Gomes, T., Pereira, J., Cabral, J., and Tavares, A. (2017). IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices. IEEE Internet Computing, 21(1):40–47.

Pinto, S., Oliveira, A., Pereira, J., Cabral, J., Monteiro, J., and Tavares, A. (2017a). Lightweight multicore virtualization architecture exploiting arm trustzone. In IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society, pages 3562–3567.

Pinto, S., Pereira, J., Gomes, T., Ekpanyapong, M., and Tavares, A. (2016a). Towards a trustzone-assisted hypervisor for real-time embedded systems. IEEE Computer Architecture Letters, 16:158–161.

Pinto, S., Pereira, J., Gomes, T., Tavares, A., and Cabral, J. (2017b). LTZVisor: TrustZone is the Key. In 29th Euromicro Conference on Real-Time Systems (ECRTS 2017), Leibniz International Proceedings in Informatics (LIPIcs).

Pinto, S. and Santos, N. (2019). Demystifying arm trustzone: A comprehensive survey. ACM Computing Surveys, 51(6):130:1–130:36.

Pinto, S., Tavares, A., and Montenegro, S. (2016b). Hypervisor for real time space applications. In The 4S Symposium.

Pinto, S., Tavares, A., and Montenegro, S. (2016c). Space and time partitioning with hardware support for space applications. In DASIA 2016-Data Systems In Aerospace.

Reinhardt, D. and Morgan, G. (2014). An Embedded Hypervisor for Safety - Relevant Automotive E/E-systems. Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems, SIES 2014.

Sabt, M., Achemlal, M., and Bouabdallah, A. (2015). Trusted execution environment: what it is, and what it is not. In 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications.

Sangorrin, D., Honda, S., and Takada, H. (2010). Dual operating system architecture for real-time embedded systems. In Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Brussels, Belgium, pages 6–15.

Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C., Anderson, A. V., Bennett, S. M., Kagi, A., Leung, F. H., and Smith, L. (2005). Intel virtualization technology. Computer, 38(5):48–56.

Varanasi, P. and Heiser, G. (2011). Hardware-supported virtualization on arm. In Proceedings of the Second Asia-Pacific Workshop on Systems, page 11. ACM.

Xilinx (2014). Xilinx UltraScale: The Next-Generation Architecture for Your Next-Generation Architecture. White Paper (Revision 1.1).

Xilinx (2016a). Zynq-7000 SoC. Technical Reference Manual, UG585 (v1.12.2).

Xilinx (2016b). Zynq Migration Guide: Zynq-7000 AP SoC to Zynq UltraScale+ MPSoC Devices. User Guide, UG1213 (v2.0).

Xilinx (2018). ZCU102 Evaluation Board. User Guide, UG1182 (v1.4).

Yalew, S. D., Maguire, G. Q., Haridi, S., and Correia, M. (2017). T2droid: A trustzone-based dynamic analyser for android applications. In Trustcom/BigDataSE/ICESS, 2017 IEEE, pages 240–247. IEEE.

Zampiva, S., Moratelli, C., and Hessel, F. (2015). A hypervisor approach with real-time support to the mips m5150 processor. In Quality Electronic Design (ISQED), 2015 16th International Symposium on. IEEE.

Zhang, N., Sun, H., Sun, K., Lou, W., and Hou, Y. T. (2016a). Cachekit: Evading memory introspection using cache incoherence. In 2016 IEEE European Symposium on Security and Privacy (EuroS&P), pages 337–352. IEEE.

Zhang, N., Sun, K., Shands, D., Lou, W., and Hou, Y. T. (2016b). Truspy: Cache side-channel information leakage from the secure world on arm devices. IACR Cryptology ePrint Archive, 2016:980.